BINARY DISASSEMBLY BLOCK COVERAGE BY SYMBOLIC EXECUTION VS. RECURSIVE DESCENT

THESIS

Jonathan D. Miller, Second Lieutenant, USAF

AFIT/GCO/ENG/12-09

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

AFIT/GCO/ENG/12-09

BINARY DISASSEMBLY BLOCK COVERAGE BY SYMBOLIC EXECUTION VS. RECURSIVE DESCENT

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Insitute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

Jonathan D. Miller, B.S.C.S.

Second Lieutenant, USAF

March 2012

AFIT/GCO/ENG/12-09

BINARY DISASSEMBLY BLOCK COVERAGE BY SYMBOLIC EXECUTION VS. RECURSIVE DESCENT

Jonathan D. Miller, B.S.C.S.
Second Lieutenant, USAF

Approved:

Dr. Rusty Baldwin, PhD (Chairman) Date

Dr. Barry Mullins, PhD (Committee Member) Date

Mr. William Kimball, M.S. (Committee Member) Date

Abstract

The increasing dependence of our nation and military on computer systems and our lack of complete knowledge about their actual capabilities is a disturbing trend. A cutting-edge technique known as symbolic execution can improve the situation in the areas of software verification and bug finding. However, there is a real need for its automated, mathematical approach in binary and malware analysis.

This research determines how appropriate symbolic execution is (given its current implementation) for binary analysis by measuring how much of an executable symbolic execution allows an analyst to reason about. Using the S2E symbolic execution engine with a built-in constraint solver (KLEE), this research measures the effectiveness of S2E on a sample of 27 Debian Linux binaries as compared to a traditional static disassembly tool, IDA Pro. Disassembly code coverage and path exploration is used as a metric for determining success. This research also explores the effectiveness of symbolic execution on packed or obfuscated samples of the same binaries to generate a model-based evaluation of success for techniques commonly employed by malware.

In its best case of uncompressed binaries, S2E achieves a maximum basic block coverage of 60% as compared to IDA Pro. It symbolically executes at least one basic block in 54% of functions and symbolically executes all basic blocks within 38% of functions. The minimum coverages are 1% basic block coverage, 3% of functions with at least one basic block symbolically executed, and 1% of functions with all basic blocks symbolically executed. Obfuscated results were much higher than expected, which lead to the discovery that S2E was not actually handling the multiple executable memory regions in unpacker runtime code. Three recommendations are made to address the shortcomings of S2E and allow it to process obfuscated samples correctly.

These results indicate that the practical limitations of S2E (and symbolic execution in general) currently outweigh its theoretical superiority as a binary analysis platform.

Selective binary analysis using this method is needed to mitigate the path explosion problem that occurs when symbolically executing an entire binary, thereby limiting use of S2E for complete binary understanding.

To the guys at 212 and 214, for their continual motivation during my time at AFIT.

Acknowledgements

I would first like to thank my advisor Dr. Rusty Baldwin for the freedom he allowed me in pursuing not only this topic but several others before this. I am also greatly indebted to Mr. Bill Kimball for his patience in explaining several complex topics in ways that were easy to understand and grasp completely, and for his constant feedback on my research. Finally, I would like to thank my research predecessor Capt. Jeff Scott for his discovery of the S2E project, continual encouragement throughout this entire research process, and very useful previous work in this domain.

Jonathan D. Miller

Table of Contents

Page

List of Figures

List of Tables

Table Page

Binary Disassembly Block Coverage by Symbolic Execution vs. Recursive Descent

1 Introduction

Computers are embedded within our most technologically advanced and expensive military weapons. They monitor and control critical infrastructure and energy grids and serve important roles in our nation's hospitals, banks, and businesses. Simply put, they are ubiquitous and irreplaceable in today's society.

As the military is charged to protect this nation's interests, it must be able to exercise dominance in the computing domain. To do this, the ability to reason correctly in this domain is critical. Reasoning means knowing all the potential functionality a device or computer system possesses. The ever increasing complexity of computer programs presents many challenges to the ability to reason.

1.1 Problem Background

The first problem with modern computing is that a high level of trust is given to others. Independent corporations, individuals, and other entities together produce and maintain the building blocks which abstract many fundamental computing problems away from the user and even the computer scientist. Compilers, hardware drivers, and operating systems are created incrementally, building on the work of previous generations and previous effort. Because of the prohibitive cost, work, and time involved in creating independent versions of these abstractions, abstraction layers largely produced outside of the control of the military are used. Thus, there are ample opportunities for a clever and malicious actor to assert control over a system during its production process, and even afterwards.

The second problem is that even the most well-intentioned programmers are still human, and so more complex software will on average contain a host of errors. A mistake in the way a program is written which results in unintended functionality is called a bug. Whether a crash is caused by a buffer overflow or a privilege is given to the wrong user, bugs are dangerous realities in software development. They may result in a software vulnerability, creating an opportunity for an attacker to exploit the code and run unauthorized programs. To make matters worse, bugs are sometimes difficult to reproduce and can depend on the individual hardware, environmental variables such as network activity, or other complicated interactions which are hard to diagnose and correct.

A third problem with complex, highly-functional systems is they are often updated. An important reason behind this constant-update mentality is to fix the bugs that have been found after the software has been shipped to eliminate vulnerabilities and unintended behavior. Another reason is to introduce new functionality. However, each time a piece of software is updated, any previously achieved understanding about the functionality and security that software offers is no longer valid. Patching or in any way updating software can introduce new bugs while trying to squash the old ones, and with them new vulnerabilities which were not present previously.

A fourth problem is one of magnitude. The path explosion problem (see Section 2.3.3) in software can lead to an exponential increase in the number of possible states in which a computer program can enter. Additionally, the memory contained within even modest laptops found in stores today can easily reach 8 gigabytes (GB), which corresponds to approximately $2^{68719476736}$ different potential states. It is infeasible to account for all the potential states a computer can enter into, and so a different method of reasoning other than exhaustively validating each state individually must be used.

To reason correctly about how computers operate, the software that runs on them must be known. This task is made substantially easier if the source code is available. With the source code, a program can be debugged as it is running, and values that the software loads into memory can be observed, along with the values assigned to variables and registers, as well as the functionality of each method. The source code can still be difficult to interpret if the program's author uses ambiguous or misleading names, values, and assignments. And, especially on large software systems, the full functionality may not be immediately apparent. However, regardless of how poorly the source code is written or if it was created to be intentionally deceiving, a program's functionality can be reasonably ascertained with this valuable information. Without the source code, the software must be disassembled into code that can be reasoned about, and this is much more difficult. Instead of requiring knowledge about software and applying reason through that domain, a reverse-engineering analyst must work backward from the program's lowest functional representation to determine the source code, and then reason about it.

1.2 Research Problem

The strategic issue is that the true functionality of a given piece of software is not well understood. Empirical evidence of this is apparent in the vibrant research areas and commercial markets of software verification, malware detection/analysis, and bug finding. Determining the true and complete functionality of any software requires the use of mathematical reasoning and deduction, and which constraint solving and symbolic execution can theoretically provide.

1.3 Research Goals

This research measures the effectiveness of symbolic execution as applied to binary analysis. It compares the disassembly generated by the industry-standard reverse-engineering software Interactive Disassembler Pro Advanced (hereafter referred to as IDA or IDA Pro) with the code translated and executed by S2E (the Selective Symbolic Execution engine) during symbolic execution of the same real-world executables. These same executables are obfuscated with a common packer, and the original executable is used as a baseline for determining the applicability of S2E and symbolic execution in general to obfuscated binaries, an area where IDA Pro and other static disassemblers have a particular problem.

1.4 Document Outline

Chapter 2 reviews the literature relevant to this research effort, and provides explanatory background information about S2E, the compilation process, and all related goals and problems. Chapter 3 presents the experiment design and implementation of both binary disassembly methods. Chapter 4 provides the results and an analysis of the experiments performed on the system under test. Chapter 5 presents the summary of research significance, avenues for future work involving this system, and research conclusions.

2 Literature Review

2.1 Introduction to Computing

Software development is a complicated process. There are several steps which must be followed to transform a programmer's source code into bits that the computer can process. Therefore, it is important to set the groundwork for how software arrives as the end product that users will execute – the compilation process itself. Literature on the decompilation process which attempts to reconstruct the original source code (or any higher-level, readable language) from machine code is presented. The decompilation process is important for it also encompasses the disassembly process which is the core of this research effort. Later sections describe decision procedures, symbolic execution, and the recursive descent disassembly algorithm. Finally, the goals of these methods and their issues are included.

2.1.1 Background.

2.1.1.1 Languages and abstraction. Assembly code is difficult for humans to read and comprehend efficiently, even the well-understood and ubiquitous complex instruction set computing (CISC) architecture of Intel's x86 platform. CISC (as opposed to reduced instruction set computing, or RISC) was originally designed as a processor instruction set with a large number of possible operations done at the assembly code level. In the earlier days of assembly code programming, these assembly instructions allowed programmers to build complex functions with fewer operations, and was seen as a vast improvement in computing since hardware design (mainly for the computer's central processing unit or CPU) was more advanced than the compilers of the time [ASU86].

The large number of functions available could theoretically increase instruction "power" for performing complex operations requiring, for instance, only one operation in contrast to the RISC approach of utilizing several simpler operations for the same functionality. However, the CISC approach may incorporate some functionality that may not be as efficient for processor hardware as several smaller operations are. The widespread presence of both platforms (albeit in different computing environments) speaks to each their important utility.

Nevertheless, CISC is more difficult for disassembly since the large number of instructions it can handle is compounded by assembly instructions which can have different byte lengths. Therefore, a disassembler must know where the first instruction ends before attempting to disassemble the next one. In contrast, the popular RISC architecture ARM has a fixed 32-bit instruction length [Ltd12], so that the start and end of each instruction is trivial to identify.

The terms RISC and CISC is slowly being overtaken by the more descriptive load-store and non-load-store, respectively. The load-store description refers to an architecture's inability to perform operations on data stored in memory directly – it must first be loaded into the CPU's registers before performing any arithmetic operations, and then stored back to memory after completion. Non-load-store architectures can retrieve data from memory, modify it, and return it to memory in a single (albeit more complex) instruction [ASU86].

Assembly code represents a program at a low level of abstraction and reveals more information about the hardware layer than data structures used in the program. The broad logic or overarching goals of a piece of software is contained at as a higher level of abstraction and is usually written in high-level languages easier for humans to understand, such as C/C++ or Java.

As compiler technology caught up to the advances in CPU technology, the prevailing notion that the assembly instruction set needed to be complex disappeared. Compilers allow the programmer to use a high-level language which abstracts away tedious and repetitive tasks necessary in assembly language programming, and allows the use of advanced data structures, common libraries, and the ability to think at a more strategic level while writing more advanced and complex software. This software is transformed via *compilation* into low-level assembly and eventually machine code for the processor to run.

2.1.2 The Compilation Process. Before a piece of software can be run by the operating system, it must be compiled to execute in the execution environment [Eil05]. This execution environment can be software-based (e.g. virtual machines) or hardware-based (with the target code natively executing on the CPU), but it requires appropriate machine code to be produced before running the software.

At its most basic level, software compilation translates a program from its original source language to a different target language [ASU86]. In most cases and in its most common usage, this means compiling an abstract high level language to a low level language. The use of compilers has allowed software developers to focus on building more advanced code in shorter amounts of time, with greater readability and understanding of the source code resulting in fewer unforeseen errors (bugs) and greater functionality.

The compilation process can be separated into three phases: the front end, mid-end, and back end. The front end contains the steps which depend on the source language. It translates them to a known intermediate language or intermediate representation (IR). The mid-end takes the intermediate representation as input and optimizes it, reducing code complexity and duplication, in a hardware independent manner. The final phase contains all necessary steps to convert the product of the mid-end, the optimized intermediate representation of the source code to machine code.

Within the front end, there are four parts: lexical analyzer, syntax analyzer, semantic analyzer, and intermediate code generator. The mid-end is composed of the code optimizer, while the back end contains the target code generator. Also present throughout all phases is a symbol-table manager, which records identifiers and their attributes, and an error handler which detects and records errors found during compilation [ASU86].

For the next sections, the stages of compilation will be illustrated using the following example program expression:

Example: *answer := base * constant + 3.5.*

2.1.2.1 Front-End. The phases of compilation allow a high degree of modularity. To compile several different programs written in several different languages on a singular architecture, theoretically all that is needed is a new front-end for each language assuming the intermediate representation provided by the mid-end can sufficiently represent all source language statements.

Lexical Analyzer. The lexical analyzer groups the characters of the source code into lexical units. These units can be identifiers, operators, punctuation, or other source language symbols. Comments within the source code are ignored [Seb99]. In the above example, the following lexical units are produced: id_1, id_2, id_3, :=, +, *, and 3.5.

Syntax Anaylzer. The syntax analyzer takes the lexical units produced and creates structures called parse trees. These parse trees are hierarchical representations of the syntax of the source code [Seb99] [ASU86]. Using the example above, the following parse tree would be created:

Semantic Analyzer. These parse trees are used by the semantic analyzer to check for errors other than syntax. It can, for example, detect if an incorrect type was used when declaring variables [Seb99]. In the example expression, if "answer" was declared as an integer instead of a floating point number or another representation capable of handling the addition of 3.5, the semantic analyzer would detect this and throw an error.

Intermediate Code Generator. The intermediate code generator produces a translation of the source program into a language that is 1) easy to produce, and 2) easy to

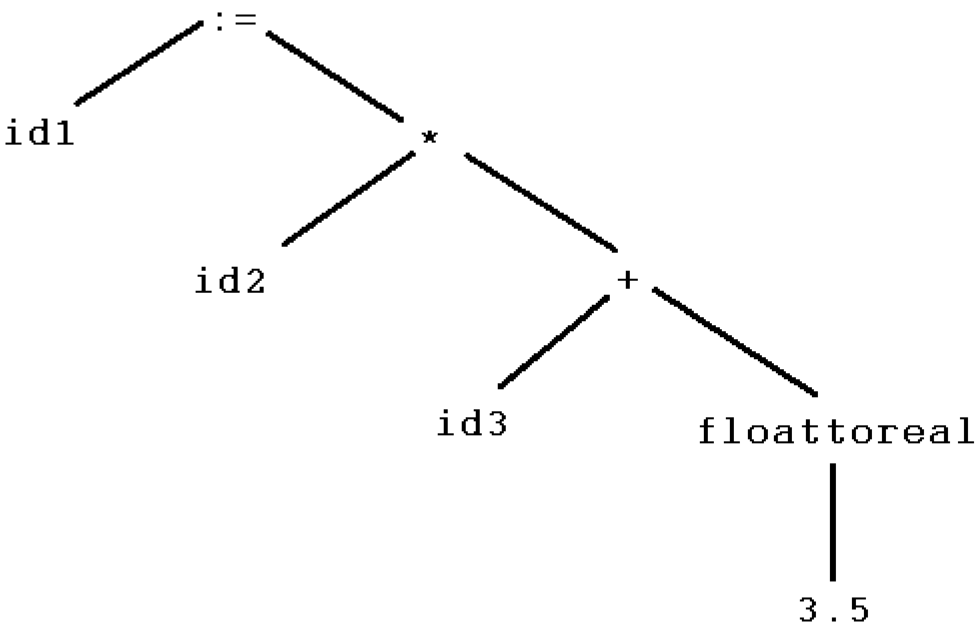


Figure 2.1: The resulting example parse tree.

translate to the target machine language. In this case, the intermediate code may look something like this:

```
temp1 := floattoreal(3.5)
temp2 := id3 + temp1
temp3 := id2 * temp2
id1 := temp3
```

2.1.2.2 Mid-End. Before optimization of code in the mid-end, a common intermediate representation used for the entire process must be agreed upon for the transformation between front end and mid-end, and also between mid-end and back end. The efficiency of the entire process (from source language to target language), as well as the final performance of the produced target code, depends on the selection of the intermediate representation. This step in the process is not "swappable" in the sense that changing the mid-end requires a change of front end and back end to accommodate any new intermediate representation.

Code Optimizer. The code optimizer attempts to improve the performance of the intermediate code when run as machine code. In the example, the code optimizer would modify the intermediate code by combining many of the assignments from the intermediate code generator into more efficient statements:

```
temp1 := id3 + 3.5
id1 := id2 * temp1
```

2.1.2.3 Back End. The back end is the final piece of the process, and unlike the mid-end is interchangeable. Given code in C, it is entirely possible to emit assembly code for different architectures as only the back end of the compiler must be switched.

Target Code Generator. The final step in the compilation process from source code to target code is the transformation from the optimized intermediate code to the target code [ASU86]. In this case, the target code is assembly language, a mnemonic version of machine code, in which names are used instead of binary codes for operations [ASU86].

```
MOV ID3, R2
ADD 3.5, R2
MOV ID2, R1
MUL R2, R1
MOV R1, ID1
```

2.1.2.4 Related Steps to Compilation. Although compilation is traditionally implemented as an independent process, other steps may be necessary.

Preprocessors Preprocessors create input for a compiler by modifying or enhancing the source program at the front end. They can process macros which extend shorthand blocks of code to longer blocks. Preprocessors fetch and include header files that are integral to the program, e.g. “#include <stdio.h>” for C. For older languages, preprocessors provide the user with newer data and flow control concepts that enhance the capabilities of an older language which may be missing them.

Loaders Loaders are responsible for two tasks at the back end of compilation. These tasks are performed on relocatable machine code. *Loading* alters the addresses of relocatable machine code to place these instructions at the proper addresses in memory while *link-editing* creates a single program by linking together relocatable machine code from several different files.

2.1.3 The Decompilation Process. Decompilers are structured in much of the same manner as compilers, with the goal of transforming the source program to another language, the target language. Compilation traditionally translates code from a high level of abstraction to a low level for execution. Therefore, it is logical to assume that decompilation is used in the opposite direction: to convert low-level code such as machine code to a higher-level language such as C++ or assembly, for examination and analysis. This process, too, is separated into three phases: the front end, the universal decompiling machine (UDM), and the back end [Cif94].

2.1.3.1 Front End. The front end of a decompiler contains machine-dependent analysis, but there are some differences compared to to regular compilers. First, there is no lexical analysis or scanning required, since all code is low-level machine language; any other level of abstraction language would use a normal compiler to group characters but in machine code there are only bytes, which could stand for data, an address, or an instruction.

Syntax Analyzer. Binary code is fed as input into the syntax analyzer, where it is structured into a hierarchical tree in a similar manner to the compiler. This structure requires machine-specific heuristics to determine its type (instruction, data, or address) of the next byte.

Semantic Analyzer. The semantic analyzer checks for semantic meaning of groups of instructions. Interpreting idioms (also known as idiomatic expressions) is a large part of this task. Idioms are entities of code which form logical operations but cannot be derived solely by considering the primary meanings of their instructions [Cif94]. For example, the idiomatic expression of shifting a register to the left or right is widely known as a shortcut for integer multiplication and division, respectively. A semantic analyzer must correctly interpret such a shift as the correct mathematical operation instead of a restructuring of its registers. There should never be semantic errors in the machine code if it was compiled, so if errors do arise they are most likely due to the semantic analyzer parsing the executable incorrectly.

Intermediate Code Generator. The semantic analyzer feeds its output to the intermediate code generator, which must create an explicit and precise representation of the source language in the chosen intermediate language. It must also be able to produce accurate translation of the source program in the target language. For most cases, a three-address code representation is preferred. This intermediate language allows three operands and can be represented by machine code and therefore a high-level language completely, without approximations [Cif94].

Control Flow Graph Generator. Once the low-level source code is represented in the IR, the control flow graph generator helps the decompiler to analyze the control flow of each subroutine. Assuming the IR is capable of representing any high-level control structures present within the program, it will attempt to achieve code length efficiency gains. For instance, in assembly language the maximum offset or distance for a conditional jump is limited because it must be mapped explicitly to memory unlike in high-level code. Often, assembly code will piggyback multiple `jmp` commands to the conditional jump so that it can be pushed further away than an explicit offset will allow. The control flow graph generator will recognize these successive intermediate jumps and set the conditional jump to the final address.

2.1.3.2 UDM. Just like the intermediate stage of the compiler, the universal decompiling machine is an interchangeable module that is independent of both source and target languages. There are some practical limitations to this theoretical approach of interoperability between several possible front-ends and back-ends, but these potential inconsistencies are dependent on the generality and expressiveness of the intermediate language used by the UDM. The UDM includes two components: a data flow analyzer and a control flow analyzer.

Data Flow Analyzer. The data flow analyzer improves the intermediate code so that high-level expressions can be found. Any reference within the IR of registers or CPU condition flags is eliminated as these concepts cannot be accessed directly by high-level languages. Thus, any assignment statements to a register are combined and assigned to the memory location from which the assignment originated. Arithmetic operations are likewise combined, using these memory locations and immediate values without including the previously-utilized registers. Other intermediate language instructions such as `push` and `pop` are also eliminated and replaced [Cif94].

Control Flow Analyzer. The control flow analyzer structures the control flow graph of every subroutine into terms of a generic high-level language [Cif94]. Only universal control flow structures are used, such as `if..then..else` statements, `for()` loops, and `while()` loops. Any language-specific implementation of control flow is not used, as it could not be faithfully translated to any target language thereby violating the modularity of the decompiler.

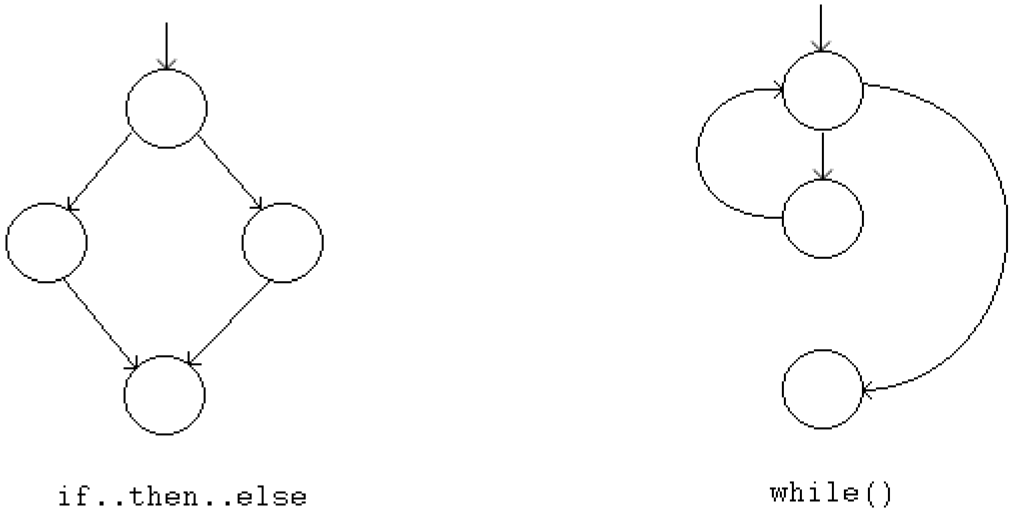


Figure 2.2: Generic Control Flow Constructs

2.1.3.3 Back End. The back end of a decompiler is the final phase in the translation to the high-level target code. When decompiling, the choice of output into several languages may give the reverse engineer additional information or insight into the program structure, or may make analysis more efficient for others who are more familiar with a different high-level language. Because of the decompiler's previously described modularity, this multiple-target decompilation is possible.

Code Generator. The final step in the decompilation process generates the target high-level code itself. The code generator accomplishes this using the control flow graph produced earlier, as well as the intermediate code of each subroutine in the program. It chooses variable names for the stack, argument, and register-variable identifiers, and selects subroutine names for each routine. Control constructs recognized in previous steps are translated to their respective high-level statements [Cif94].

2.1.4 Disassembly. Perfect decompilation of a binary to the original source language undoubtedly offers the most straightforward method for analysis. However, this is impossible with current techniques. Disassembly is widely viewed as a more manageable problem, but due to the intermingling of data with instructions, there is (in CISC architectures such as x86) the real possibility of having more than one plausible disassembly. Disassembly therefore reduces to the Halting Problem and is not solvable in general [HM79]. Several static disassembly algorithms have been proposed for most efficiently solving this problem, and they are described below.

2.1.4.1 Linear Sweep Disassembly. Linear sweep disassembly is by far the most straightforward, though its use in variable-length instruction sets like x86 is not recommended. Linear sweep begins disassembly at the first byte in the code section and moves linearly through the section, one instruction at a time until the entire code section is complete. It keeps track of a pointer at the beginning of each instruction which allows it to determine the current instruction length. This is helpful for fixed length instruction sets (e.g. MIPS, ARM). It does not attempt to account for control flow, but it does provide complete coverage of the program's code sections. It cannot account for data that is sometimes comingled with code, and so in variable-length instruction sets it may confuse a piece of data for a new instruction. This type of disassembly is provided by GNU debugger (`gdb`), Microsoft's WinDbg, and the Linux `objdump` tool [Eag08].

2.1.4.2 Recursive Descent Disassembly. Recursive descent disassembly focuses on the concept of control flow and determines whether or not an instruction should be disassembled on the basis of references from other instructions. This is easy with simple sequential flow instructions like `add`, `mov`, `push`, and `pop`, and linear sweep is used in these cases. In cases where control flow branches to a known point (a direct branch), such as the x86 `jnz` (jump if not zero) instruction, both branches of the conditional will be disassembled. If a conditional depends on runtime values, such as x86 `jmp eax` requiring a jump to the value stored in register EAX, these values cannot be computed in static disassembly. Such a jump (or `call eax` or `ret`) depend on the values of registers or the stack at runtime and are referred to as indirect branches. In these cases, IDA Pro and other static disassembly methods that use recursive descent cannot properly continue disassembly, and must halt. Whenever a series of instructions is finished disassembling or can no longer be followed, the algorithm chooses a new series of instructions to continue disassembling, giving this method its recursive characteristic [Eag08]. Recursive disassembly may not always provide complete disassembly due to the presence of indirect branches, but its ability to distinguish code from data by following the control flow of a program is a distinct advantage over linear sweep.

2.1.5 Application of Formal Logic to Software. There are two distinct methods of reasoning through the rigorous basis of formal logic. The model-theoretic or “Model Checking” approach enumerates all possible solutions from a finite number of candidates. The proof-theoretic or “Theorem Proving” approach uses a deductive mechanism of reasoning based on axioms and inference rules, which when used together form an inference system [KS08]. These two methods of reasoning are the basis for propositional logic and provide mathematical soundness and completeness.

When reasoning about computer systems, several considerations must be made. If the number of system states is finite, Model Checking can be used and the problem is decidable. However, in real systems, the number of possible machine states with respect to input is astronomical, on the order of 2^n where n is the number of memory address bits. For 8 GB memory as is ound in many laptops, the number of possible states becomes $2^{68719476736}$, a number larger than the number of atoms in the universe. This is important because it means that checking each of these states becomes infeasible, and a different strategy must be used to formally verify these systems.

First-order logic solves this problem by ordering all possible states into sets, and reasoning about these sets of states. Symbolic execution is an extension of first-order logic; it is a sequence of state transitions represented by Single Static Assignment (SSA) form. SSA form is a widely used representation for compiler optimization and greatly simplifies dataflow optimizations [Lat02]. A program is in SSA form "if each variable is defined exactly once and each use of a variable is dominated by that variable's definition [Lat02]." As explained in Section 2.1.5.4, Low-Level Virtual Machine (LLVM) is a suitable choice for the intermediate representation (IR) of a symbolic execution engine largely because of its adherence to SSA form.

2.1.5.1 Constraint Solving. Satisfiability is a fundamental problem in computer science. It measures whether a formula expressing a constraint can produce a solution, and is found in hardware and software verification, type inference, test-case generation, and other problems [dMB09]. Propositional satisfiability (SAT) is the most popular form of constraint satisfiability problem. The goal of SAT is to decide whether a formula created from Boolean variables and logical connectives can be made `true` by choosing Boolean `true`/`false` values for its variables [dMB09].

At the most basic level, this research is most interested in binary analysis – the ability to determine exactly what functionality a given binary executable is capable of without access to its source code. To make this decision, the program must be represented as a series of mathematical statements. These statements are most often expressed as formulas which use first-order logical connectives, variables, quantifiers, functions and predicate symbols [dMB09]. Satisfiability Modulo Theories (SMT) problems are examples of these logical first-order formulas and are also known as decision problems.

2.1.5.2 Decision Procedures. A decision procedure is an algorithm that when given a decision problem, returns with a simple (and correct) yes/no answer. If yes, the decision procedure will also return a satisfying assignment [KS08]. Decision procedures have traditionally been used in hardware verification and analysis, for testing paths through a circuit and verifying its functionality correctly matches the specification through each path.

Decision procedures can also be used for the logic of computer software. Advances in the decision procedures for bit-vectors (the binary method in which values and data are stored on computers) and arrays (an important data structure for organizing this data) have led to their adoption in automatic theorem provers and SAT solvers [KS08][GD07]. The theories considered most promising in the bit-vector and array decision procedures are still difficult; the problem of deciding them is NP-complete [KS08].

2.1.5.3 Symbolic Execution. Symbolic execution is an automated approach to testing and analysis based on the paths found through a program. It substitutes program input with symbolic values and manipulates these instead of performing traditional operations on a given input. Values are generated to satisfy symbolic constraints generated during the execution path, which are then solved by SMT solvers [KHL10] [FCN+10]. All possible paths through a program are taken and symbolized until path termination or a bug is found, generating a test case for the condition found.

This approach is limited given the sheer number of possible paths that will need to be represented by the symbolic execution tool. These possibilities are generated by the exponential number of paths through which execution could flow, and the entropy introduced by any environment the program must run within (user, network, hardware, etc). The former problem, known as path explosion (see Section 2.3.3), can be mitigated but not eliminated with symbolic search heuristics and other methods to determine which paths are more likely than others. Other approaches include limiting symbolic execution to a more reasonable domain.

2.1.5.4 LLVM. The Low Level Virtual Machine, or LLVM, is a compilation framework based upon the LLVM intermediate representation (IR). The LLVM IR offers a limited degree of architecture independence for several languages and is the language to which a source language is optimized before the transformation to the machine code of a specific architecture. LLVM [Lat11][Lat02] is widely utilized in a modular compiler architecture, with the ability to serve as the mid-end for languages such as C, Ada, C++, and even ARM assembly. Previous research such as ABACAS (the Architecture-independent Binary Abstracting Code Analysis System) [Sco11] focused on translating ARM assembly to LLVM using traditional disassembly techniques.

From this mid-end, LLVM can be compiled to run on architectures ranging from Intel's x86 to ARM to server-specific Itanium processors. Along with its flexibility, LLVM also brings impressive performance, matching and even outperforming GCC in compile time as well as execution speed [KLMK10].

2.2 Goals of Disassembly and Symbolic Execution

The most important question to ask when covering reverse engineering is "Why reverse?" Improvements to reverse engineering techniques and advancements in understanding are useful. But there are several practical reasons for achieving a high degree of understanding in these areas.

2.2.1 Offensive. The first major reason for reverse engineering is to manipulate code. When a bug is found in code special exploits can be crafted to take advantage of it, whether it be in the operating system or an application. Studying the inner workings of Microsoft Windows or Adobe Reader has yielded thousands of malicious viruses, botnets, and Trojan horses, specifically targeted to attack certain configurations, yet widespread enough to cause millions of dollars in productivity and infrastructure damage [Eil05]. In the right hands, this knowledge creates a unique opportunity for intelligence gathering and a method to impose our will upon other organizations or countries in a non-violent manner.

2.2.2 Defensive. This information can also be very dangerous in the wrong hands. Therefore, it is equally important to apply expertise to the defense of computers, networks, and intellectual property by finding unexpected flaws and faults in the software so they can be quickly and comprehensively patched before disclosure to the public (and by default the hacking community), thus saving computer systems from potential exploitation. It is equally important to identify and detect malicious code on a computer or computer network. Through reverse engineering, patterns emerge in malicious programs that can be used as signatures [Eil05].

2.2.3 Understanding. Reversing also helps programmers learn from the work of others. By examining the methods used to solve a particularly advanced problem, students become experts themselves. This is a very useful method of learning, and helps develop a strong base of code knowledge [Eil05]. Understanding code also discovers new potential features that the original developers never considered. The creativity that comes from new eyes on a piece of code can contribute to new and innovate opportunities for the original program.

2.2.4 Future-Proofing. The majority of reversing work happens on x86, with relatively little on the new ARM architecture. Mobile computing is a newer market, and thus reverse-engineering software is not available on all traditional platforms. It is of the utmost importance, though, to establish and cultivate experts in fields that have a strong potential for growth and adoption, to mitigate potential threats. Mobile computing has proven itself to be one of these explosive areas, and thus the familiarity with ARM-based reversing will enable future offensive and defensive capabilities understanding.

2.3 Issues with Disassembly and Symbolic Execution

Although the perfect recreation of source code from machine code would be optimal, there are limits to what can actually be obtained. The compilation process is destructive, obliterating comments, redundancies, and other structures that could be useful to analysts. There are some techniques that can recover the original meaning and functionality of the code, despite often-imperfect results.

2.3.1 Data Type Inference. A common problem in reverse engineering is the abstraction of data types originally present in source code are stripped during the compilation process. Inferring these types is known as type reconstruction. This reconstruction is extremely helpful to an analyst but difficult because of optimizations made to map data to registers and the global memory of the system. There are two parts to this problem: variable recovery and type recovery [LAB11].

2.3.1.1 Variable Recovery. Variable recovery identifies the high-level source code variables from low-level machine code [LAB11]. This is straightforward as each new parameter passed to a function is accessed via `ebp` offsets, and each unique offset (e.g., 0xC and 0x10) signifies the existence of a unique parameter.

2.3.1.2 Type Recovery. Type recovery associates a high-level data type to each variable. This is much more challenging because these types are usually discarded by the compiler. Solutions have been proposed in both the commercial and research fields for this problem.

IDA Pro. Hex-Ray's IDA Pro [HR11], for instance, includes a disassembler which uses proprietary heuristics to infer a type. However, these heuristics are often inaccurate, classifying `unsigned int` and `unsigned int *` as `int`. This misidentification of signed data types appears to be IDA Pro's default behavior [LAB11].

REWARDS. Another approach called REWARDS [LZX10] follows a single path of execution using dynamic analysis. The restriction of the analysis to a single path was made in anticipation of the difficulty of determining and keeping track of a program's control flow [LZX10][LAB11]. REWARDS uses information from executed "type sinks" which are common function calls with known types. For instance, a call to `strlen` would mean that the type of this function's argument is a `const char *`, since this is the only type that `strlen` will accept. Any assignment from this argument to another variable means that this new variable has the same `const char *` type.

However, REWARDS cannot handle control flow and therefore is unable to perform the commonly-utilized reverse engineering method of static analysis. Additionally, its approach cannot be run multiple times dynamically since that would require control flow, which is determined via static analysis [LAB11].

TIE. A new inference-based approach, TIE (for Type Inference on Executables) uses several "hints" or indicators about the way the code is used. For example, TIE checks the signed flag to infer whether two operands are signed in an arithmetic operation [LAB11]. From these hints, TIE builds a series of formulas that are solved revealing data types more conservatively and accurately than REWARDS or IDA Pro. TIE lifts the assembly code to binary analysis language, then uses variable recovery to identify potential variables in the code. It then builds its formula-based constraints, based on aforementioned hints. It solves the constraints to the greatest resolution possible, leading to both precise and conservative results [LAB11].

2.3.2 Data Structure Recovery. A useful high-level language incorporates data structures such as strings, arrays, lists, stacks, trees, and queues. These differ from data types in that they cannot be manipulated directly from assembly language. Data structures are collections of primitive data types such as characters, integers, and floating point numbers. During compilation, these structures are obliterated as they are stored in memory, assigned as the compiler sees fit to increase the speed of execution.

2.3.2.1 Static Analysis. Some approaches to data structure recovery use locations in binary that appear to be variables, and then estimate the values they hold. This

Value Set Analysis (VSA) uses interpretation to give a best estimate of sets of values [SSB11]. Its algorithm tracks both addresses and integers simultaneously, creating over-approximations of both addresses and the set of integer values that could be held at each point in the program [BR04]. The actual locations are difficult to identify explicitly in these interpretations, so another technique called Abstract Structure Identification (ASI) observes the manner in which memory is accessed by the program. Consider a particularly large segment set aside for local variables with a short, few-byte access made midway through the segment. ASI would conclude that the segment is made of a short few-byte variable, surrounded by two larger arrays. It also uses the type signatures of well-known functions to determine their types automatically in a similar manner to REWARDS but during static analysis rather than dynamic [SSB11].

CodeSurfer/x86. Both VSA and ASI are used in an experimental IDA Pro plug-in called CodeSurfer/x86 [Res11][BR04]. This tool can browse, inspect and analyze x86 executables and recovers IRs from these executables as if they were retrieved midway through the compilation process.

2.3.2.2 Dynamic Analysis. Unlike static analysis, dynamic analysis only inspects code that will be run. Three popular tools for doing this are Laika, REWARDS, and Howard.

Laika. Introduced as a tool to analyze various self-modifying viruses, Laika fixates on the idea that internal data structures are reused throughout code. It avoids code polymorphism present in samples and instead detects botnets through their use of the same internal data structures [CSXK08]. By identifying potential pointers within code and the structures that follow them. If, for instance, the structure (up until the next pointer is observed) ends in a null-termination and is a series of ASCII characters, it is probably a string. It uses this logic for all kinds of potential data structures. However, its detection method is imprecise because it does not detect the fields in the structures, making Laika unsuitable for reverse engineering [SSB11].

REWARDS. The aforementioned similarity between ASI and REWARDS is used in data structure recovery to identify well-known functions. Unfortunately, REWARDS is incapable of detecting and identifying internal data structures or structures that are created and used exclusively by a particular program [SSB11].

Howard. A more recent approach to data structure recovery borrows from many of these ideas and is named Howard. Howard uses clues from pointer memory access to infer the layout of data in memory. Pointers are used for all memory access, whether it be through direct addressing or indirectly through a CPU's registers [SSB11]. For example, if a program accesses memory at point `A`, then `*(A + 4)` is probably an element in this structure. If the structure is an `int []` array, it has accessed the second value in the array [SSB11].

2.3.3 The Path Explosion Problem. Using symbolic execution engines fails to scale to large programs because the number of execution paths that must be symbolically considered is so large that only a very small portion of the program's entire execution tree can actually be reached. This problem is caused mainly by the presence of loops, conditionals, and nested calls in the program and therefore is a difficult problem for modern software analysis [KHL10]. These high-level programming constructs cause the number of paths to increase exponentially as the program executes, which then requires symbolic execution to represent and solve an exponentially increasing number of program states [CDE08].

2.3.4 Obfuscation. Software obfuscation attempts to hide the true functionality of a program, preventing analysis and understanding by entities which do not possess the source code. One of the most common types of obfuscation, packing, is designed to bypass detection or code fingerprinting by most antivirus tools, and is also very easy to apply to compiled code. The remainder of this section addresses packers and the techniques employed which make packed executables very difficult to detect.

2.3.4.1 Packing. Packing is a method originally designed to compress a binary program so that it occupies less disk space, preventing potential issues with moving large binaries. However, malicious usage of this technique has made detection of malicious software (or malware) difficult. The packing concept is illustrated in Figure 2.3.

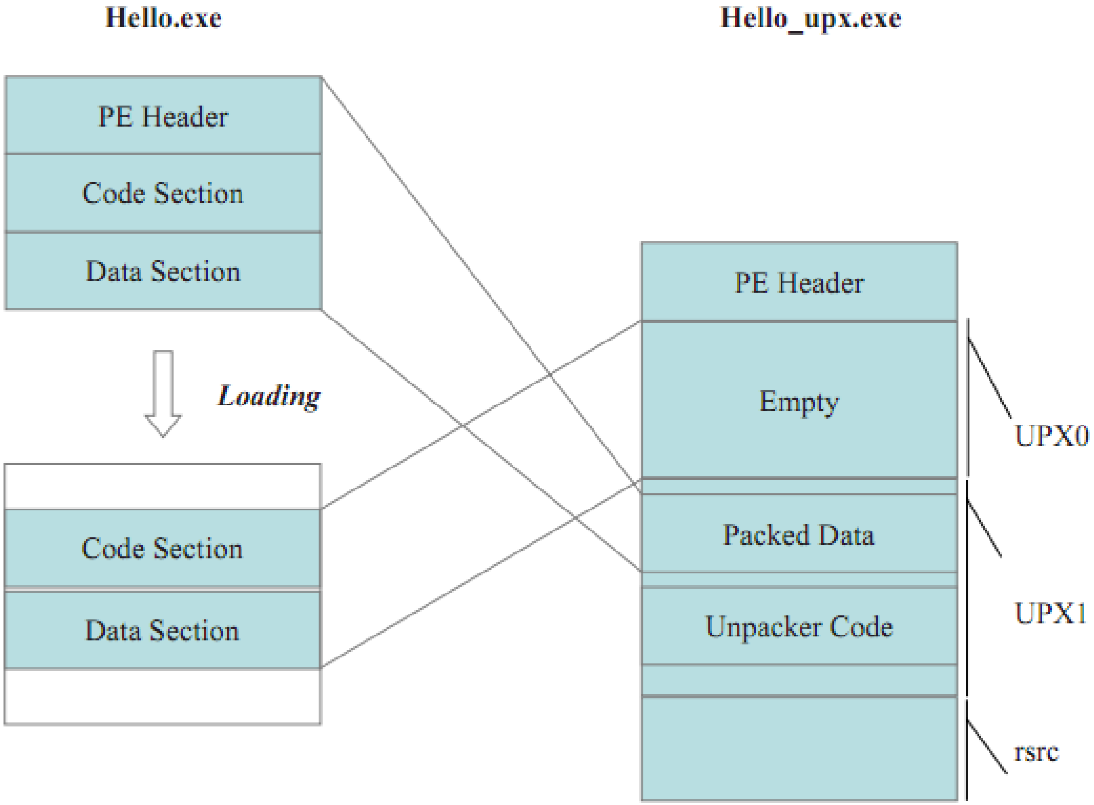


Figure 2.3: The inner workings of UPX (Universal Packer for eXecutables), a common packing tool, on a Windows binary. Image source: GFC08

First, the executable is packed. This process combines the program header, code section, and data section into the Packed Data section of the new binary. Compression techniques allow a substantial reduction in disk space usage for modest-sized programs [OMR12]. The new packed (or compressed) program consists of a new header, a virtual address space which adequately holds the content from the original binary's code and data sections (UPX0), the packed data itself, and unpacker code (both in UPX1). For resources that cannot be compressed, an extra resource (rsrc) section is added. The new header points directly to the unpacker code, skipping the empty sections where the real functionality of the program is loaded.

When the new compressed binary is run, the program header launches the unpacker code first. The unpacker loads the code and data sections into the virtual address space, UPX0, created earlier during the packing process. This code is functionally equivalent to the uncompressed version, but does not have the same signature. Static disassembly tools such as IDA Pro would show the unpacker code, unaware of where and how the original binary would be loaded into memory.

2.3.4.2 Techniques. Compressing an executable with a packer like UPX introduces an added level of complexity for binary analysis. But with a method for detecting the packer and the version used, unpacking the software becomes easier. For this reason, many malware authors add several layers of packing to their software, requiring the use of a series of unpacking tools in the correct order to reach the real binary. A general expression for multilayer packing of an executable is

$$X \xrightarrow{F} F(X) \xrightarrow{G} G(F(X)) \tag{2.1}$$

where *X* is the executable, *F(X)* is the binary packed with the *F* packer, and *G(F(X))* is *F(X)* after being compressed with packer *G*. The recovery of the original binary from this multilayer packing is

$$G(F(X)) \xrightarrow{G'} F(X) \xrightarrow{F'} X \tag{2.2}$$

where *G’* is the unpacking option of packer *G*, and *F’* is the unpacking option of packer *F*. To recover the original executable, it must be unpacked first by *G’* and then by *F’*. The unpacking operation is not a commutative operation, and therefore must be performed in reverse order of the binary that was packed.

Compounding the difficulty is a malware author’s use of a proprietary, closed-source, or custom packing tool. In this case, there is no official method for extracting the original code, it would require a just-in-time antivirus scanning tool such as Justin [GFcC08] to determine if the file is contains previously-identified malicious software. Malware analysis on a binary compressed within a custom packer would require manual debugging with a tool like OllyDbg [Yus11], and extend efforts to determine even basic functionality of the program.

2.4 Summary

Reverse engineering is an increasingly important aspect of protecting and securing networks and data. By reversing the usual compilation process of a executable program to produce disassembly, analysts can then more easily observe its execution, and therefore determine its functionality. This derived understanding can be used to patch vulnerabilities in software to prevent attacks that exploit these vulnerabilities. The information gleaned from reverse engineering is also a valuable learning tool as it can reveal methods used by a sophisticated piece of malware, allowing security experts to recognize and stop similar attacks from damaging our computer systems before they occur.

3 Experimental Methodology

Reverse engineering of binaries is a relatively mature albeit manual process for traditional computing platforms such as the Intel x86. However, it is also extremely labor-intensive and highly dependent on an analyst's proficiency and can require several tools to accomplish adequately. S2E provides an automatic, single interface for binary path exploration and analysis.

3.1 Problem Definition

3.1.1 Goals and Hypothesis. The overall goal of this research effort is to determine the effectiveness of symbolic execution as a method for binary analysis. Basic block and function block code coverage of a binary's disassembly based on symbolic execution versus industry-standard static disassembly tools is the primary metric. This effort uses the S2E Selective Symbolic Execution Engine [CKC11][CKC12] and a plugin built for that framework called `init_env` that analyzes even proprietary and closed-source Linux binaries so the disassembly effectiveness can be evaluated over a range of obfuscation levels. Binaries at several levels of obfuscation are analyzed as obfuscation presents a particular problem for static disassemblers such as IDA Pro.

It is expected that block coverage achieved during symbolic execution will remain constant over increasing obfuscation levels. It is also expected that the block coverage of static disassembly will be high on unmodified binaries, but drop dramatically as obfuscation is introduced. The results of this research will determine whether utilizing symbolic execution will expose more functionality than static disassembly techniques on an obfuscated binary. This chapter describes the methodology for evaluating the S2E system and that the research goals have been met.

3.1.2 Approach. The research hypothesis is tested by submitting a variety of binaries to static disassembly as well as symbolic execution. Each binary is obfuscated at different levels using the Ultimate Packer for eXecutables (UPX), and re-run through both disassembly methods. After symbolic execution is complete, offline tools generate the different execution paths found, complete with PC address forks taken by S2E to create new states, and the PC addresses of each translation block executed.

Using the number of basic blocks found by IDA Pro as a baseline for uncompressed binaries, provides a high degree of assurance that the entire state tree has been exhausted and thus that all potential paths have been exposed. This meets the research goal of determining whether selective symbolic execution is an effective alternative to static disassembly relative to current methods of analysis.

3.2 System Boundaries

The System Under Test (SUT) is the S2E selective symbolic execution framework. S2E consists of a customized virtual machine, a dynamic binary translator based on QEMU [Bel11], an embedded symbolic execution engine [CKC11][CKC12], a plugin architecture with several example plugins, and a suite of post-execution analysis tools. An detailed overview of the individual components which make up S2E is presented in Figure 3.1.

The workload submitted to S2E is comprised of several levels of compressed binaries. These are individually submitted to S2E and selected during runtime with the `init_env.so` library which overrides the start of the function to insert symbolic execution boundaries around the binary as it is loaded into memory [CKC11][CW12]. These boundaries are interpreted by the S2E-modified QEMU system emulator, which translates the code within the boundaries to LLVM for later analysis by KLEE, or to the host's native machine code for efficient execution. As the code is executed in either the

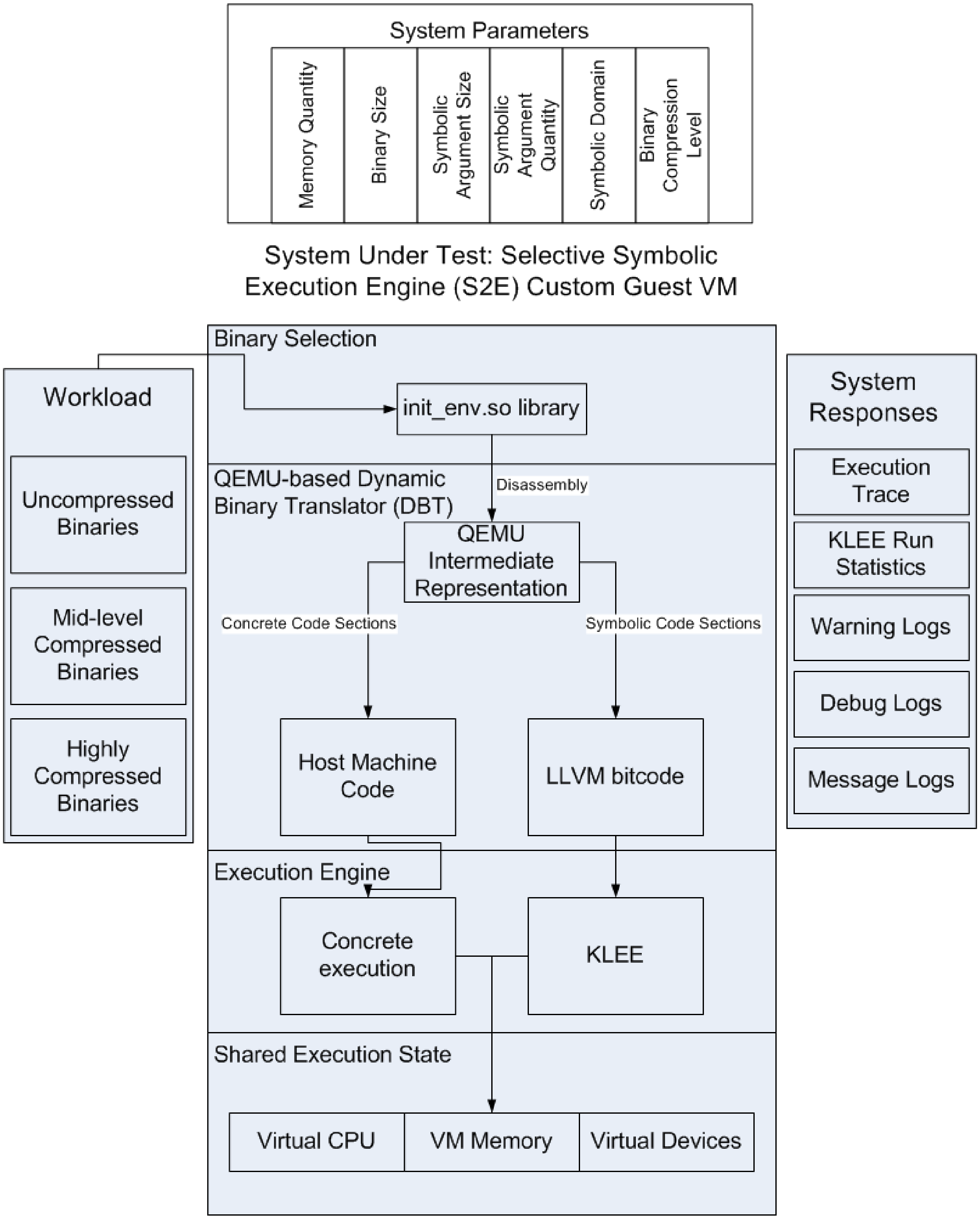


Figure 3.1: System Under Test: The S2E Selective Symbolic Execution Framework.

symbolic or actual domains, new states are created for new execution paths. These states are shared and updated as either execution method progresses. After S2E has finished (either due to finding all the possible states of a program, or by running out of memory), it outputs a series of files for use by post-execution analysis tools [CKC11][CKC12].

This research is limited to x86 binaries on Linux which accept input from the command line using the `init_env` plugin as a driver for the process. The instruction set is limited to x86 because S2E does not currently offer official support for the ARM architecture, though experimental modules are under development for this purpose [CCKK12]. Since `init_env` is the only available plugin which allows Linux binaries to be symbolically executed without prior instrumentation and it currently supports only command line arguments, the testing setup is limited to the command line.

3.3 System Services

S2E service is the symbolic execution of a binary, with an output of its complete execution trace. Ideally, the system symbolically executes any x86 program flawlessly with complete functional equivalence, but environmental variables (e.g., network activity, system state, the user) and theoretical limitations (e.g., path explosion) can introduce too many paths for the system to execute within a feasible amount of time. Within the service of selective symbolic execution, there are several sub-services: selection of code to be run natively or symbolically, dynamic translation to LLVM for symbolic execution, symbolic execution itself, statistical run generation, and coverage generation.

For efficient and accurate decompilation, an appropriate target intermediate language must be available. The Low-Level Virtual Machine (LLVM) IR has emerged as a capable mid-end target with a precise compiler infrastructure built around it.

3.3.1 Dynamic Binary Translation. This service is provided by the S2E-modified version of QEMU. Previous research has used the QEMU system emulator to dynamically translate x86 machine code to LLVM during execution [CC10]. With the appropriate front end, this research can be extended to any machine code language that LLVM can represent. The entire symbolic execution framework is dependent on this Dynamic Binary Translator (DBT) to offer an appropriate level of selectivity, separating the program of interest from system libraries and operating system code so that only the former is symbolically executed. To accomplish this, S2E defines a series of S2E-specific commands that are interpreted solely by the DBT: `s2e_enable_forking` and `s2e_disable_forking`. These two commands mark the beginning and end, respectively, of the code section sent to KLEE for symbolic execution instead of being natively executed by x86 hardware.

Earlier versions of S2E required the explicit inclusion of these commands within source code; the `init_env` plugin (later included with S2E source as a guest library) changed that requirement by placing them dynamically around the program at run-time. These commands appear to the compiler as processor opcodes, but are in fact unused and therefore ignored during compilation, only to be interpreted by the modified QEMU as code that should be passed to KLEE for symbolic execution.

3.3.2 Branch Concretization. This service is necessary to explore different branches along a code path. When KLEE reaches a condition, it must take one of the possible paths for execution to continue. Branch concretization changes a symbolic argument to a concrete one that meets the condition for taking the branch. When this condition is met, the branch is taken, and S2E forks the entire execution state with an S2E-specific instruction. Processor register values, contents in memory, and the current condition taken are saved so that when switching to this new state S2E knows to take the alternate branch instead.

3.3.3 Branch Selection. This service is provided by KLEE. Its path selection mode is set at random and therefore may not always result in the same number of paths per unit time. The individual branch taken at a conditional is random in the sense that there is no set algorithm (e.g., breadth-first or depth-first) that must be followed [CDE08].

3.3.4 Dynamic State Translation. This service preserves the entire execution state as values are changed from symbolic to concrete and vice versa by S2E's manipulation of the state object which is shared and then updated between KLEE and QEMU. This service is extremely important to preserve the selective nature of S2E's symbolic execution; current values for all methods must be constantly maintained in either the symbolic or concrete domain to continue execution. For example, if a function is part of the program being symbolically executed, and the program calls a outside library to perform some operation on a value used by the function, it must be converted to an appropriate value in the the concrete domain, and then returned to the symbolic domain once the library has finished processing it [CKC11][CKC12].

3.3.5 Constraint Solving. Once translated to LLVM, the KLEE symbolic execution engine takes over, running the code marked for symbolic execution through Simple Theorem Prover (STP), a constraint solver. STP is optimized for bit-vector arithmetic and arrays, two of the most important ideas in basic data storage in computing which are also common bottlenecks in software verification and bug-finding applications [GD07].

3.3.6 Statistical Run Generation. This service is provided by KLEE. As each binary is symbolically executed, KLEE reports the number of current states, translation blocks (concrete and submitted to KLEE), CPU instructions, time spent in user mode, querying, solving, and forking, and memory usage among other things. This information is important for determining the performance of KLEE itself and where any performance bottlenecks occur. Successfully generating these run statistics imply success in all other previous services since this service depends on those.

3.3.7 Coverage Generation. To analyze S2E's basic block code coverage the binary is loaded and stored onto the QEMU guest's disk image. Then, S2E-modified QEMU is run with the binary selected and any options and plugins desired. As the binary is symbolically executed, S2E creates and maintains an execution trace file (`ExecutionTrace.dat`) along with several log files about the run. Separately, the same binary must be opened with IDA Pro and analyzed with the `extractBasicBlocks.py` script, included in Appendix B.2. The post-execution `coverage` tool is then run, and requires the `ExecutionTracer.dat` file, `.bblist` file, and the executable itself. By comparing these basic blocks to those generated by IDA Pro, the total code coverage is obtained. This is dependent on success in all other services except statistical run generation.

3.4 Workload

The workload for the system is a collection of three sets of files. The first is the x86-based binary files compiled for and bundled within the 32-bit Debian/GNU Linux 5.0 distribution (see Section 3.10). These real-world binaries provides assurance that the results are not purely theoretical but instead reflect potential performance for future usage on unknown binaries.

The second and third sets are comprised of the first set, but obfuscated using the UPX packer at levels `-5` and `--best`. The submission of workload is not a function of time; only a single binary can be submitted at a time.

3.5 Performance Metrics

3.5.1 Measured Performance. The following performance metrics measure the effectiveness of the S2E system on native (uncompressed) binaries.

- Basic Block Coverage (%BBCov): The percentage of all disassembled basic blocks symbolically executed by S2E. A basic block is defined as a section (block) of code that has exactly one entry point and one exit point. Therefore, once the first statement in the basic block is executed, all other statements in that block will by definition be executed. Basic block coverage is an absolute measurement of the amount of code covered in comparison to the basic blocks discovered by IDA Pro.

$$\%BBCov = \frac{CoveredBasicBlocks}{TotalDisassembledBasicBlocks} \tag{3.1}$$

- Touched Function Coverage (%TFCov): The percentage of all functions within the binary which are at least partially symbolically executed by S2E (at least one basic block). Touched function coverage reveals how many of the functions within the code were actually invoked during symbolic execution, and measures code coverage in as many areas of the binary as possible. This is also calculated using IDA Pro as a baseline.

$$\%TFCov = \frac{TouchedFunctions}{TotalFunctions} \tag{3.2}$$

- Fully Explored Function Coverage (%FEFCov): The percentage of all functions within the binary in which all basic blocks are symbolically executed by S2E. Fully Explored Function Coverage measures how many functions within the code achieved 100% execution, that is, it measures complete coverage of the maximal amount of functions throughout the binary. This uses data collected from IDA Pro to determine if each function has been completely executed.

$$\%FEFCov = \frac{FullyExploredFunctions}{TotalFunctions} \tag{3.3}$$

- Number of Paths/States Discovered: The number of paths discovered while symbolically executing each of the binaries. A higher number of paths suggest path explosion was not a significant issue as paths are short enough to terminate before memory is depleted.

3.5.2 Model Performance. Since S2E can still run compressed binaries (albeit without all loaded memory addresses), we can obtain run statistics from KLEE to compare against their native counterparts and estimate performance when coverage support for obfuscated binaries is added in the future.

- Number of Paths/States Discovered: The total number of states found in the compressed binary. It is expected to be lower since the uncompressed binary will have a greater percentage of the actual code loaded into memory where S2E expects it to. Whereas the uncompressed binary, on the other hand, will most likely have only a small proportion of the same code, if any at all.

  With this metric, intelligent predictions can be made about the coverage potential of the symbolic execution engine once it can measure coverage without the input from IDA Pro. Since the coverage tool currently requires IDA Pro to compare basic/function blocks with the S2E output, the other metrics used for the uncompressed binaries cannot be applied.

3.6 System Parameters

The following parameters affect the system performance.

- Memory Quantity: The amount and type of Random Access Memory (RAM) available to the system. More RAM will allow S2E to hold more states during symbolic execution and therefore increase the total symbolic execution time of the system (assuming the path explosion problem persists and the number of possible states exceeds the amount of memory that can be dedicated to the operation). Increased memory will also allow more block coverage, since more paths can be explored.

- Binary Size: The compiled size of the program to be tested, measured in kilobytes (kB). Larger programs will likely have more execution branches, leading to a higher number of states and longer execution time. It is expected that the number of blocks found will decrease as the size of the program increases (keeping all other parameters constant) since only so many paths can be found with limited resources, and this relatively similar number of paths will make up a smaller proportion of a larger program.

- Symbolic Argument Size: The maximum size in bytes of each symbolic argument that can be fed through the command line to each binary in the workload. Larger sizes are expected to open more potential paths in the binary for symbolic execution than a smaller number, leading to more basic block coverage. However, it is also expected that an increase in argument size will mean more memory usage, and since S2E is already memory-constrained, this may decrease the total number of actual paths found.

- Symbolic Argument Quantity: The range of possible symbolic arguments that can be fed through the command line to each binary in the workload. This is represented as a range of numbers, from 0 to x, where x is the maximum number of arguments. Similar to symbolic argument size, the number of these arguments may expose more paths to explore, but may not actually be explored due to increased memory load of each state needing to store more arguments.

- Symbolic Domain: The executable memory address space available for symbolic execution in S2E. This address space may be 1) for executing only code belonging to the process itself (–select-process-code), 2) to include other code in user address space such as libraries (–select-process-user), 3) to include other code in all address space, including kernel space (–select-process). The user may limit symbolic execution only to the code contained within the executable itself and not worry about library calls or higher-order OS code to be included within the symbolic realm. It is expected that using only the process address space will limit the potential number of paths that can be found. This, however, is a desirable trait as then expensive symbolic execution cycles are not spent exploring and creating OS-level states.

- Binary Compression Level: The level at which the UPX packing tool has compressed each of the binaries for addition to the workload. Compression levels range from 1 to 9 and "best," which offers the most compressed executable possible. Higher compression leads to more memory consumption during runtime decompression and execution. Thus, higher compression is expected to result in decreased block coverage and shorter symbolic execution time.

3.7 Factors

Two parameters are selected as factors to test the performance of S2E. Their corresponding levels are described below and summarized in Table 3.1

Binary Size: The effect of an executable's size plays on block coverage cannot be ignored. Since the sizes of the original 27 binaries are in some cases have a two orders of magnitude difference, they were separated into four classes: <25 kB, 25-50 kB, 51-100 kB, and >100 kB. This classification introduces some context to potential differences in block coverage and execution time.

Compression Level: To maintain a representative sampling of the disassembly performance of S2E for multiple compression levels each executable was run at 1) a minimal level of compression which in this case is uncompressed (none), 2) medium level of compression (`upx -5`) and 3) maximum level of compression (`upx --best`).

Table 3.1: Experimental Factors and their Respective Levels

Levels	Binary Size	Compression Level
1	<25 kB	none
2	25 - 50 kB	-5
3	51 - 100 kB	–best
4	>100 kB	

Although there are four levels for the Binary Size factor, there are not an equal number of binaries available for each of these classification levels. Therefore, a full factorial experiment is run using the total number of binaries at each of the three compression levels. Sufficient statistical basis for analysis is expected to be achieved with one replication. Thus, the research will have 27 * 3 = 81 total experiments.

3.8 Evaluation Technique

A hybrid approach to evaluate the system is used, utilizing both measurement and modeling techniques. Measurements are taken from the S2E system itself on uncompressed binaries. For these samples, measurement is the evaluation technique because the system is configured to accept binaries in this form and already generates block coverage based on the output from IDA Pro. Additionally, the time and resources required to measure this portion of the workload is no greater than those needed to model the experiments. Therefore, measurement of the current system is preferred.

To evaluate the system's potential future effectiveness in block coverage of obfuscated binaries, a modeling approach is used. Preliminary testing on obfuscated binaries reveals that S2E cannot provide block coverage statistics for two reasons. First, it depends on the successful output from IDA Pro to generate the `.bblist` file, matching PC addresses to basic blocks, and these basic blocks to their functions. Without successful output from IDA Pro, there is no standard on which to base the discovered basic blocks and other metrics such as functions covered. Conceivably, this same problem could manifest itself even for non-packed and non-malicious files if the disassembly from IDA Pro is incorrect.

The second reason obfuscated binaries are unable to be measured is that they cannot be properly run on S2E. When S2E detects a binary running, it is configured to monitor the memory addresses of that binary so S2E can report which addresses have been executed. Currently the `init_env` plugin can only track modules with contiguous, unchanging memory space. The run-time decompressing an obfuscated binary undergoes during its execution uses a different amount of memory than S2E accounts for and also loads program data into several locations in contrast to a normal executable's contiguous, single address space. Modifications to the `init_env` plugin to recognize new "modules" when the program allocates new memory would solve this problem.

However, the obfuscated samples are still symbolically executing the code immediately within its assumed memory addresses, so the measurements that KLEE makes during symbolic execution still occur. Using these measured values will contribute to the obfuscated samples' model-based evaluation.

Each of the binaries is tested by running them within S2E. For the native, non-obfuscated binaries, the performance of S2E is evaluated using equal weighting from the measured performance metrics from Section 3.5.1. For the obfuscated binaries, the performance (or, more accurately, the potential future performance) of S2E is evaluated using model performance metrics from Section 3.5.2.

3.9 Testing Phases

Several phases of testing are used to determine the effect of various factors on experimental results, and accommodate the increasingly deeper understanding of how S2E accomplishes symbolic execution. Different factors were chosen in each phase of testing to separate and understand the effects that parameters had on the system.

3.9.1 Phase I. The first phase bases its factors and levels on analysis of the S2E options available. Only one replication is used for each executable since the results are being compared to a single run of the binary through IDA Pro. For this phase, the parameters of symbolic argument size and symbolic argument quantity are chosen. The number and size of the arguments submitted to the test binary is specified as a range (e.g., from zero to five paths of up to four bytes each is specified as `--sym-args 0 5 4`). Phase I determines whether these parameters should be included in the final experiment design.

Several levels of argument size and quantity are tested on the `echo` binary. Since S2E developers constantly referenced `echo` as an example executable in their tutorials, it is assumed that its relative simplicity allows it to perform consistently for this initial testing. As Table 3.2 shows, more paths were discovered as the number of arguments and their sizes are increased. It is assumed that a greater number of paths leads to greater block coverage by S2E.

Several levels of argument size and quantity are submitted to S2E, with the largest of them producing the highest number of states found and the most paths through the program. Therefore, a large quantity of potentially large-sized arguments (e.g., up to 10 arguments, each of size 12 bytes, or `--sym-args 0 10 12`) would include all of the possible subsets of smaller argument quantities and smaller argument sizes.

Table 3.2: Effect of Arguments Given on Number of Paths Found

		Argument Quantity Range	
		0 to 2	**0 to 10**
Argument Size (bytes)	**4**	8158	11914
	12	10128	12498

3.9.2 Phase II. The second phase of testing analyzes the basic block and function block coverage with the offline analysis tool included with S2E, `coverage`. It also used the `run.stats` file to determine the cause of runs seeming to end prematurely. The results are surprising: each time a run had "completed," the operating system had actually run out of memory, and killed S2E. The path explosion problem manifested itself even on `echo` and it seemed that the initial testbed which used 16 GB of RAM was actually part of the problem. With that large amount of memory, S2E could keep track of thousands of new program states and continue to search for new paths for far longer than it found new basic blocks. In fact, the last time S2E found a new basic block in `echo` was 311 seconds into the experiment; the next 3 hours and 25 minutes were spent unsuccessfully searching the execution paths for a new basic block (see Table 3.3).

Table 3.3: Effect of RAM Size on basic block coverage on the `echo` binary. `Echo --sym-args 0 10 12` was used in this phase.

Size of RAM (in GB)	Basic Block Coverage (%)	Exact Blocks Covered
3	5.038	53 of 1052
16	5.228	55 of 1052

Running the same binary with only 3 GB of RAM results in the same block coverage, since the last successfully found basic block occurred before the number of states grew too large for the VM to hold. Therefore, using `echo` as a pilot reduces the amount of memory previously thought necessary, and allows multiple experiments to run simultaneously in separate VMs. Since `echo` is a very simplistic program and yet still suffers from path explosion, it is assumed that the other binaries in the workload will also see decreased test times at no expense to the block coverage achieved using the same decreased RAM levels. In effect, reducing the available memory shortens the wasted time due to path explosion.

The creation of and adherence to these phases maximizes the potential for discovering new relevant information, minimizes data which is redundant and/or unnecessary, and leads to efficient experimentation procedures.

3.10 Experimental Design

To evaluate the interaction between all factors, a full factorial design is used. There are two factors: executable size and obfuscation level. Executable size is obtained using the `objdump` Linux command line utility and is measured in kilobytes (kB). Obfuscation level is determined by the UPX packing program [OMR12] UPX version 3.08 (released 12 December 2011). UPX has 10 different available compression levels, from 1 to 9 and "best," which can take significantly longer but has the best compression ratio [OMR12]. Levels "5" and "best" were used as representatives of moderate and "extreme" compression, respectively.

For this research, a suitable Linux disk image is loaded for QEMU (and the S2E back end) to run. Debian GNU/Linux 5.0 "Lenny" is used, as it had been released four years prior (on 6 February 2008). It is therefore considered a mature release of Linux. Since the `init_env` plugin requires executables be driven from the command line, a non-graphical release of Linux is preferred to reduce overhead from possible graphical elements. Therefore, this disk image includes support for command line operation only. The 32-bit version of Lenny is used, simplifying the pool of potential executables that undergo analysis and excluding their 64-bit counterparts. Additionally, the OS image is compiled for x86 hardware, and must be of specific image format (qcow2, or QEMU Copy On Write version 2). This disk image format delays allocation of storage until necessary and also supports snapshots of QEMU [Bel11], which is important for the experiment setup. Since QEMU is a full system emulator, it takes a significant amount of time (on the order of several minutes) to start, especially considering it is operating within the S2E-provided Ubuntu VM running on VMWare Workstation. These snapshots access the same system state in seconds from both the original, unmodified QEMU emulator and the S2E-QEMU emulator.

On the Debian GNU/Linux 5.0 distribution, there are 83 executable files located in the standard Linux `/bin/` directory, which together comprise the Coreutils suite of command-line tools available on every Linux distribution. However, there are many which are not used. First, many of these executables are duplicates (e.g. `nc` and `netcat`). In these cases, the binary with the more descriptive name is used. Second, some of the binaries offer only a subset of another's functionality (e.g., `rnano` is a restricted version of `nano`, and `fgrep` is simply the `grep` command with the `-f` flag enabled). When this occurs, the more functional binary is used. Third, many of the executable files are not actually binary files, but instead are shell scripts (e.g., `uncompress`, `which`, `zdiff`). Finally, S2E cannot symbolically execute everything. Each executable was initially analyzed in a pilot test with S2E to determine whether the tool could work. Of the 83 original files in `/bin/`, 30 meet the previous requirements and are able to successfully execute within S2E; of those, 27 can be further compressed successfully by UPX for a comparison of handling obfuscation levels. The list of binary executables used and a description of each from their respective `man` (manual) page can be found in Appendix A.

The testbed setup is provided in Figure 3.2. This figure provides context for the information contained within the rest of this section.

Each run is completed within a customized version of the S2E-provided VMWare virtual machine which runs the Linux-based operating system Ubuntu 10.10 (available at https://s2e.epfl.ch/attachments/download/63/s2e-demo-vmware.tar.bz2). S2E itself is

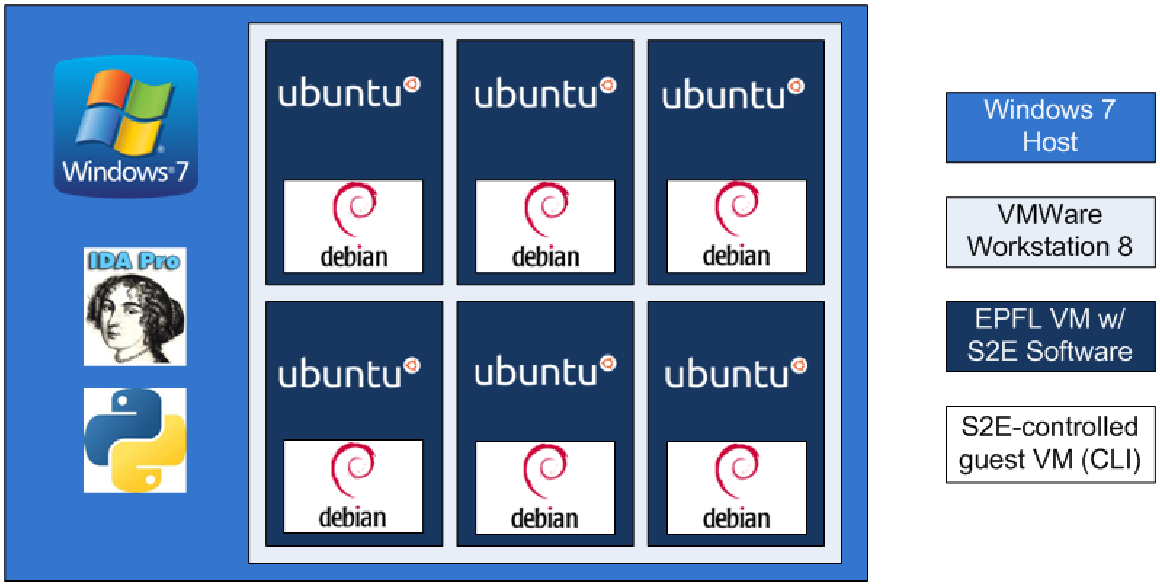


Figure 3.2: The testbed setup for this research.

hosted on a `git` repository (at https://dslabgit.epfl.ch/git/s2e/s2e.git); S2E commit c3822c82 from 6 Feb 2012 is used in all experiments. To increase execution speed (and using the knowledge gained from dedicating alternatively larger and smaller amounts of RAM to the VM), each of 6 VMs is run on 3 GB RAM, each with one dedicated processing core from the host machine. The host machine runs on Windows 7 64-bit with Service Pack 1, and has 24 GB RAM and two quad-core Intel Xeon X5677 CPUs running at 3.47 GHz. Running six S2E VMs simultaneously leaves 2 dedicated cores and 6 GB RAM for the host. This is more than adequate considering the only extraneous host process running during testing is VMWare Workstation (version 8.0.1 build-528992) itself.

Each of these experiments is compared to the output achieved in IDA Pro with the same executable. IDA Pro Advanced version 5.7.0.935 (32-bit) was used, with support for detecting and analyzing the Linux ELF executable binary file format [HR11]. In addition, IDAPython version 1.4.2 is installed [Sou12] to support Python 2.5.4 scripts [Com08]. Once disassembled, a custom Python script bundled with the S2E source code named `extractBasicBlocks.py` (see Appendix B.2) extracts the addresses and related functions of the basic blocks found in the executable. This was stored as a `.bblist` file, which is transferred back to the Ubuntu VMs for analysis with the included `coverage` tool.

The `coverage` tool bundled with S2E compares the trace of symbolic execution (named `ExecutionTrace.dat`) with the `.bblist` file generated from IDA Pro, and produces `.repcov`, `.timecov`, and `.bbcov` files. These files are used with the `run.stats` file from the initial symbolic execution generated by KLEE to determine the relative coverage achieved by S2E in comparison to IDA Pro and the time required for each stage of symbolic execution.

3.11 Methodology Summary

The decompilation of x86-based binaries is an important technique for comprehension, vulnerability analysis, and bug correction of closed-source code. A crucial step in decompilation is the disassembly of machine code into an intermediate representation which is subjected to constraint-solving. The expected result from this research is that disassembly focusing on symbolic execution will outperform current methods of recursive descent disassembly as obfuscation levels increase.

This experiment symbolically executes a representative sample of relatively small binaries as well as two obfuscated versions of each in S2E. They are also be disassembled in IDA Pro and the resulting block coverage compared.

4 Experimental Results and Analysis

The performance metrics described in Section 3.5 characterize the performance of S2E for binaries across several compression levels. Section 4.1 presents the performance of native, uncompressed binaries. The results for obfuscated binaries are in Section 4.2.

4.1 Non-obfuscated Results

Measurement of block and function coverage for non-obfuscated (also referred to as native or uncompressed) binaries extracted from the Debian guest system serves as baseline performance. From this baseline, some idea is possible of how well the obfuscated versions of these binaries could expect to be covered if IDA Pro produced accurate `.bblist` files. The metric maximum states is the reference for symbolic execution through the binary for later comparison to obfuscated samples.

4.1.1 Metric: Maximum States Found. Each of the binaries is symbolically executed in S2E as described in Section 3.10. Before running the `coverage` tool, the number of paths are determined with `tbtrace`, the Translation Block Tracer tool. As expected, the quantity of paths found with tbtrace were equal to the number of states found by KLEE. This is expected since each path should result in the creation of another execution state by S2E. The state quantities are in `run.stats`, a file generated by KLEE and placed in the `s2e-last` folder with all other data gathered by S2E from that run. Figure 4.1 shows the total number of states found in each binary during symbolic execution.

Two very distinct patterns emerge from these non-obfuscated binaries and therefore they will be discussed separately. Section 4.1.1.1 interprets the results of those binaries which produced very few states (18 or less), while Section 4.1.1.3 does the same for those binaries with hundreds more states.

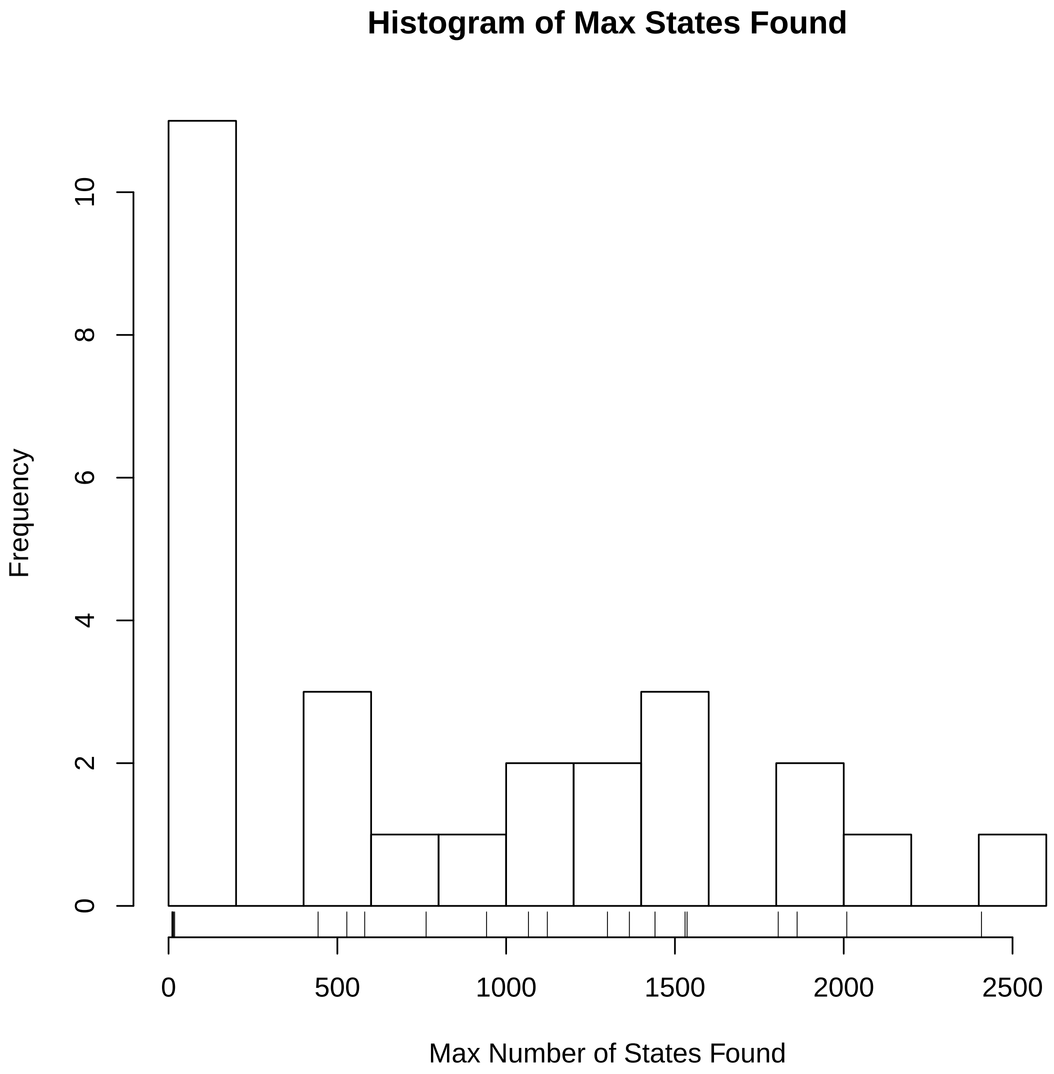
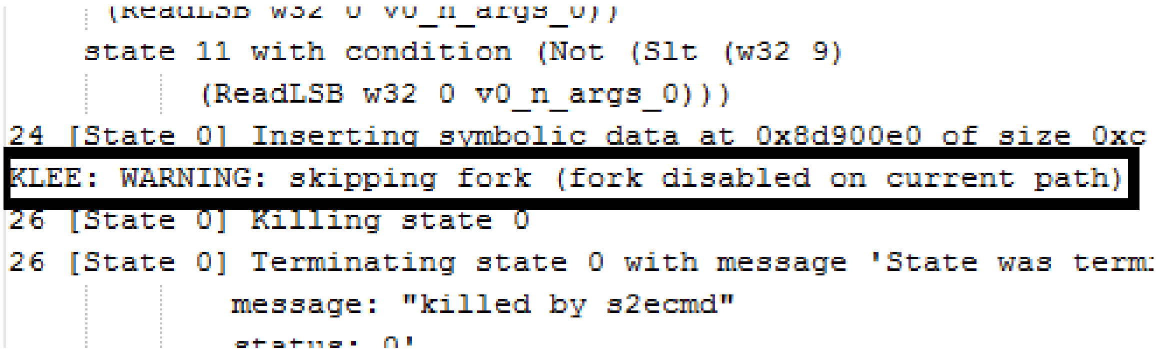


Figure 4.1: Histogram of number of states found for all non-obfuscated binaries with actual data points along x-axis.

4.1.1.1 Observation I: Few Paths Found. For 11 of the 27 native samples, fewer than 20 total paths were found through the execution tree. As shown in Figure 4.1, they are concentrated on the far left side, between the ranges of 11 and 18 total states in what appear to be a single value in the figure. Intuition dictates that complex binaries such as `busybox` with a large amount of functionality (see Appendix A.2) cannot contain only

11 possible execution paths. Further analysis of execution logs also reveals a warning in KLEE common to all of these abbreviated state trees, shown in Figure 4.2.



```
      (ReadLSB w32 0 v0_n_args_0))
    state 11 with condition (Not (Slt (w32 9)
          (ReadLSB w32 0 v0_n_args_0)))
24 [State 0] Inserting symbolic data at 0x8d900e0 of size 0xc
KLEE: WARNING: skipping fork (fork disabled on current path)
26 [State 0] Killing state 0
26 [State 0] Terminating state 0 with message 'State was term:
            message: "killed by s2ecmd"
            status: 0'
```

Figure 4.2: The common warning present in the output generated during binaries with few paths found.

This warning means forking (or symbolic execution) has been disabled for the remainder of the execution tree. Along with a significantly lower number of paths, these binaries were not constrained by memory like those with hundreds of paths found. Instead, the relatively low number of paths terminated quickly after discovery. Additionally, an interesting phenomenon occurred in this subset of the workload. There were more repeated instances of the same maximum states (11 states were observed six times) than any other amount.

4.1.1.2 Interpretation I: Few Paths Found. The results from this subset are unexpected but can be explained. Since many binaries use outside libraries to parse arguments from the command line, these libraries are not loaded into the same address space as the rest of the binary's code. This feature of S2E allows symbolic execution to proceed selectively enough for the analysis of individual binaries. Thus, the same number of symbolic arguments (from 0 to 10) is passed to the binary but since it must use external library code to parse the arguments, the rest of that branch is concretized (i.e., forking is disabled), and upon return to the binary's code section is killed.

4.1.1.3 Observation II: Many Paths Found. The remaining 16 uncompressed samples explored at least 443 execution paths with some finding over 2000. For each of these binaries, the `run.stats` file indicated that the VM had allotted all its available memory to S2E before killing the process entirely to recover memory. During the symbolic execution of `bash`, for example, the last reported update to the `run.stats` file showed 3446153216 bytes of memory usage, or approximately 3.4 GB. This corresponds with the 3 GB allocated to the VM, along with 480 MB used as swap space.

4.1.1.4 Interpretation II: Many Paths Found. The relative success of symbolic execution finding hundreds or thousands more paths than the first subset of the non-obfuscated binaries shows the true potential of S2E as a binary analysis platform and the theoretical superiority of symbolic execution as an automated binary analysis method. However, path explosion limits the true number of paths S2E can explore and other limitations such as usable memory and time are apparent.

4.1.2 Metric: Basic Block Coverage. Basic block coverage is an important metric to establish an absolute value of what percentage of all executable code is actually submitted to KLEE for symbolic.

4.1.2.1 Observation. The basic block coverage achieved by S2E varied greatly for this workload. However, no relationship could be established to predict the coverage of a binary with any other previously-gained information such as binary size, number of states found or ratio of size to states found. Several scatterplots (cf. Appendix C.1) show no clear relationship with respect to these variables. Thus, this information is presented in a boxplot shown in Figure 4.3.

Overall basic block coverage was low in comparison to IDA Pro. The maximum basic block coverage is in `mktemp` at 60%. This is much higher than the interquartile range between 8% and 29% but not so much that is considered an outlier. The lowest

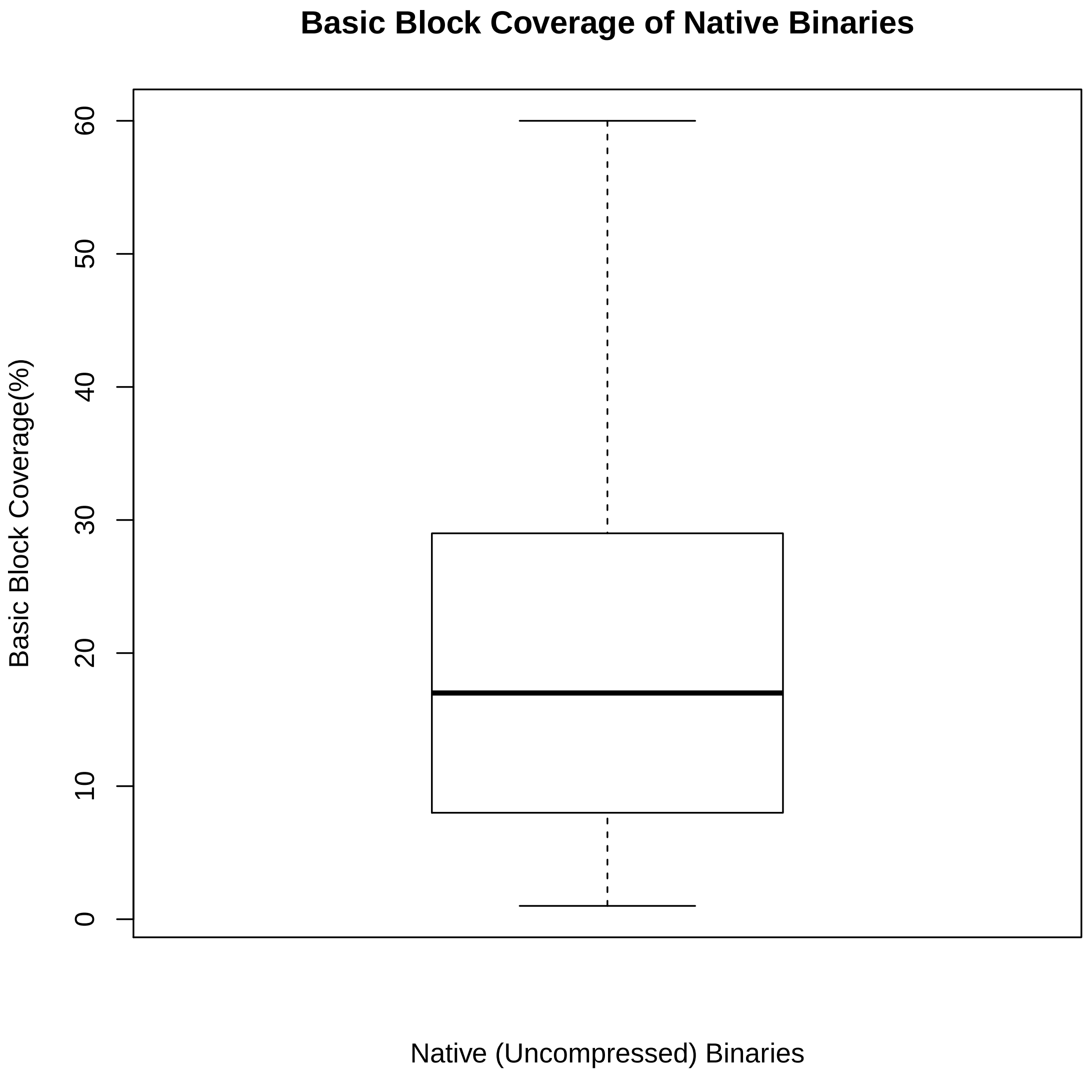


Figure 4.3: Basic block coverage of native (uncompressed) binaries.

basic block coverage was on `busybox` (cf. Appendix A.2), a suite of reduced-functionality UNIX commands in which S2E could only find 11 paths despite its relatively large size (368 kB, the second-largest in the workload).

4.1.2.2 Interpretation. The complexity of these binaries was by far the most important characteristic with respect to S2E's ability to achieve high basic block coverage.

For example, `mktemp` and `busybox` had quite disparate coverage percentages. Viewing these binaries in IDA Pro through its function call graph viewer was a revealing exercise for why these coverages are so different. The `mktemp` graph is shown in Figure 4.4, while Figure 4.5 presents the same for `busybox`.

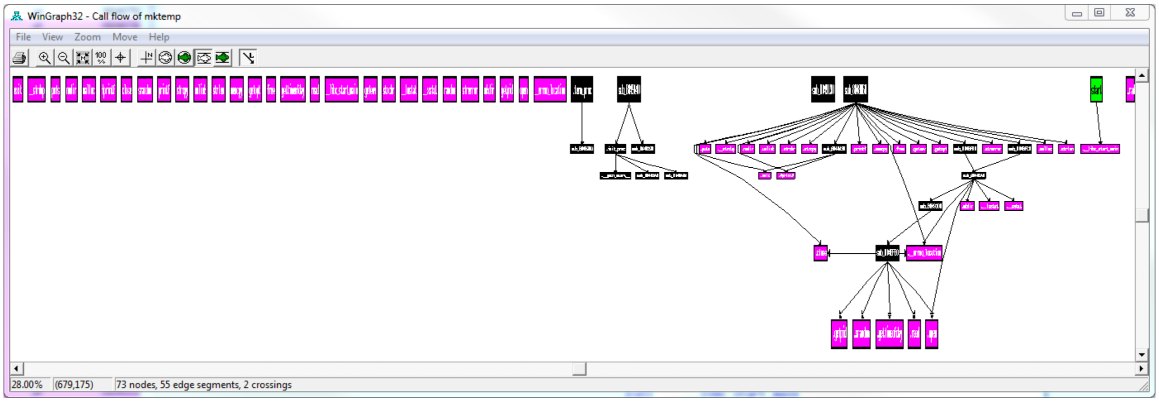

WinGraph32 - Call flow of mktemp
File View Zoom Move Help
28.00%
(679,175)
73 nodes, 55 edge segments, 2 crossings

Figure 4.4: Function call graph produced by IDA Pro for `mktemp`.

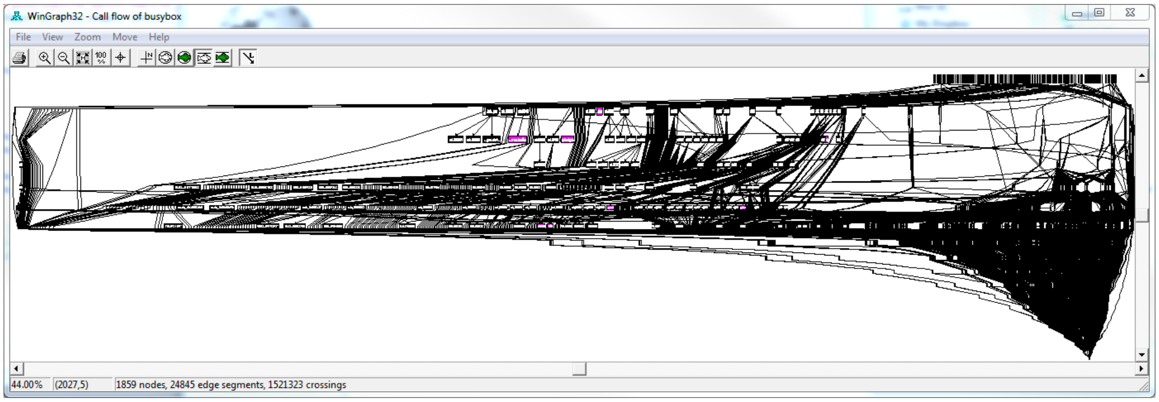

WinGraph32 - Call flow of busybox
File View Zoom Move Help
44.00%
(2027,5)
1859 nodes, 24845 edge segments, 1521323 crossings

Figure 4.5: Function call graph produced by IDA Pro for `busybox`.

The lighter pink boxes are named functions while black boxes represent internal, unnamed functions. Calls from one function to another are represented by black arrows. Clearly, `mktemp` has a much smaller and simpler program structure which allows greater coverage of its basic blocks. The number of branches present in `busybox` are so numerous that they appear to be solid and its larger size requires the graph to be sacled substantially compared to `mktemp`.

In addition, `busybox` only resulted in the discovery of 11 total paths. This suggests that its method for parsing input is located in an external library, unable to be reached by S2E for symbolic execution as it is outside of its allocated memory space.

4.1.3 Metric: Touched Function Coverage. The touched function coverage measures how effectively S2E can provide at least shallow coverage of as many functions as possible. This is important because the more functions covered, the greater potential that command-line arguments have in exploring the program. Symbolic execution is inherently limited by the path explosion problem, but success in finding at least the beginning of each function (and later failing to finish because of real-world memory or time constraints of representing the exponentially increasing number of execution states) portends the potential of using S2E or other CLI-based methods in driving path exploration.

4.1.3.1 Observation. As with basic block coverage, it was difficult to establish a relationship between touched function coverage and other previously-known information (cf. Appendix C.2). Figure 4.6 presents the touched function coverage of all native binaries.

Touched function coverage was extremely varied ranging from 3% with `busybox` to 54% with `sed`. The median value was 31%, with an interquartile range of 22% to 40%.

4.1.3.2 Interpretation. The coverage for `sed` is comparatively high suggesting that it relies mostly on user-driven input from the command line to achieve most of the functionality present in its execution tree. Its description as a stream-based command line text editor supports this conclusion (cf. Appendix A.23 for its detailed `man` page description). Again, `busybox` has low coverage, due to S2E's inability to fork on any

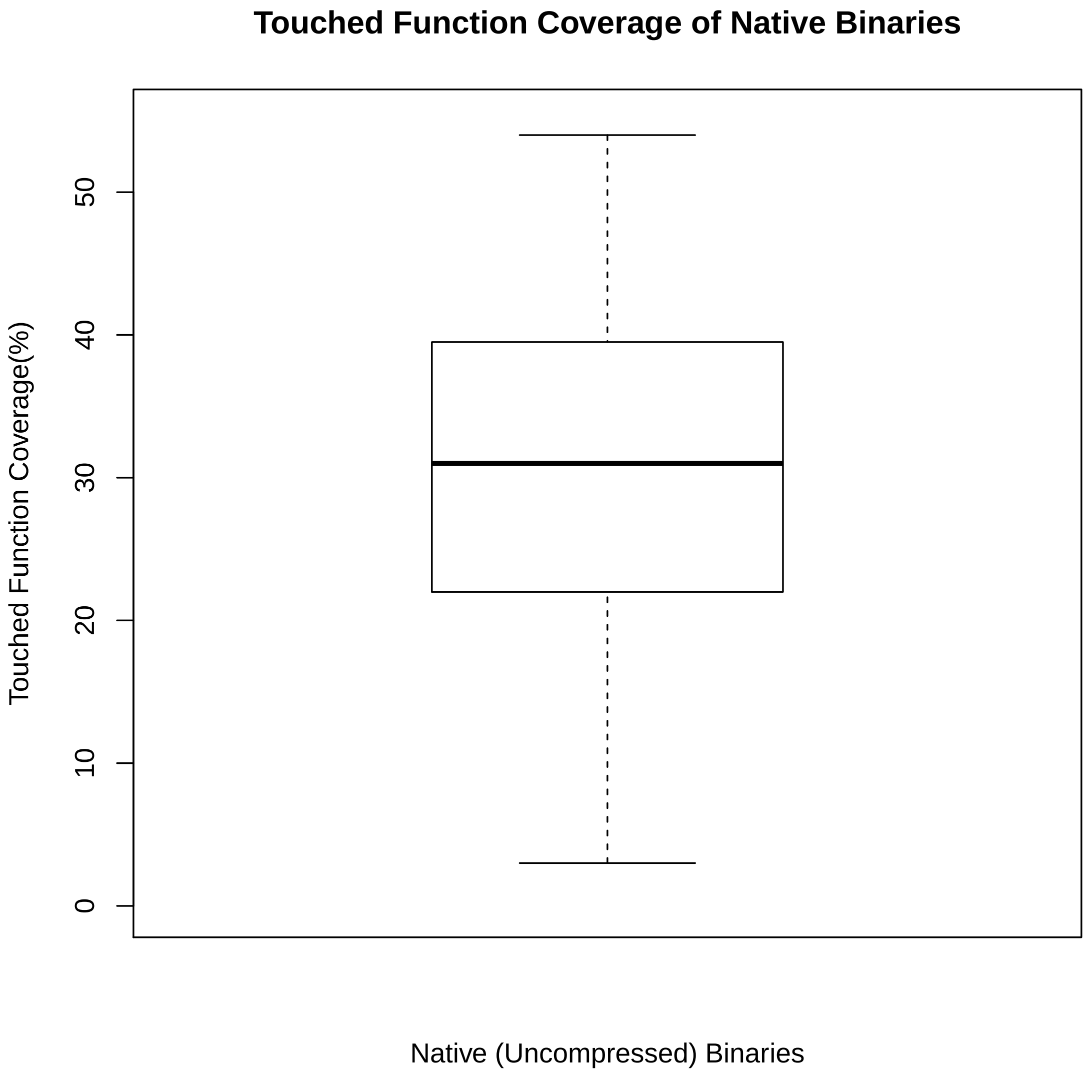


Figure 4.6: Touched function coverage of native (uncompressed) binaries.

code that is outside of the memory space allotted to it. This prevented `busybox` from using the symbolic input provided and therefore resulted in disappointing performance.

4.1.4 Metric: Fully Explored Function Coverage. Each function in a program is comprised of a number of basic blocks. The fully explored function coverage measures the percentage of functions for which S2E symbolically executes all basic blocks. That is, all code which can be considered part of that function is executed. This is a measurement of the shallowness of a program – that is, it may have many paths, but some of them are so simple that symbolic execution can reach their termination point. The termination point must be reached before S2E is killed by the operating system to regain memory.

4.1.4.1 Observation. It appears estimating future symbolic execution performance of an executable is very difficult without some preliminary analysis with IDA Pro (cf. Appendix C.3). Size, states found, and even ratios of the size and states fail to provide any apparent correlation with respect to coverage.

The maximum fully explored function coverage was during the run of `date` at 38%. The minimum coverage was again in `busybox` at 1%. The median coverage was 22%, with an interquartile range of 14.5% to 27%.

4.1.4.2 Interpretation. The relatively high fully explored function coverage for `date` can be attributed to the simplicity of its paths and the ability for S2E to keep a higher number of states active while searching through them until termination. A prime example of this simplicity is the amount of time `date` was able to run without using all of the system's memory. Even though it is a 54 kB binary (well above the median 33 kB), it ran for the second-longest time (61429 sec, approximately 17 hours) because many of its paths were simple enough to find an end, freeing up memory for additional states and executions paths. For the third time, `busybox` achieved the lowest coverage due to S2E preventing it from parsing and therefore using the symbolic input.

4.2 Obfuscated Results

As described in Section 3.5.2, different metrics for obfuscated binaries needed to be appropriately chosen given S2E coverage tool's reliance on accurate data from IDA Pro, and IDA's failure to provide meaningful results on obfuscated samples. Therefore, the

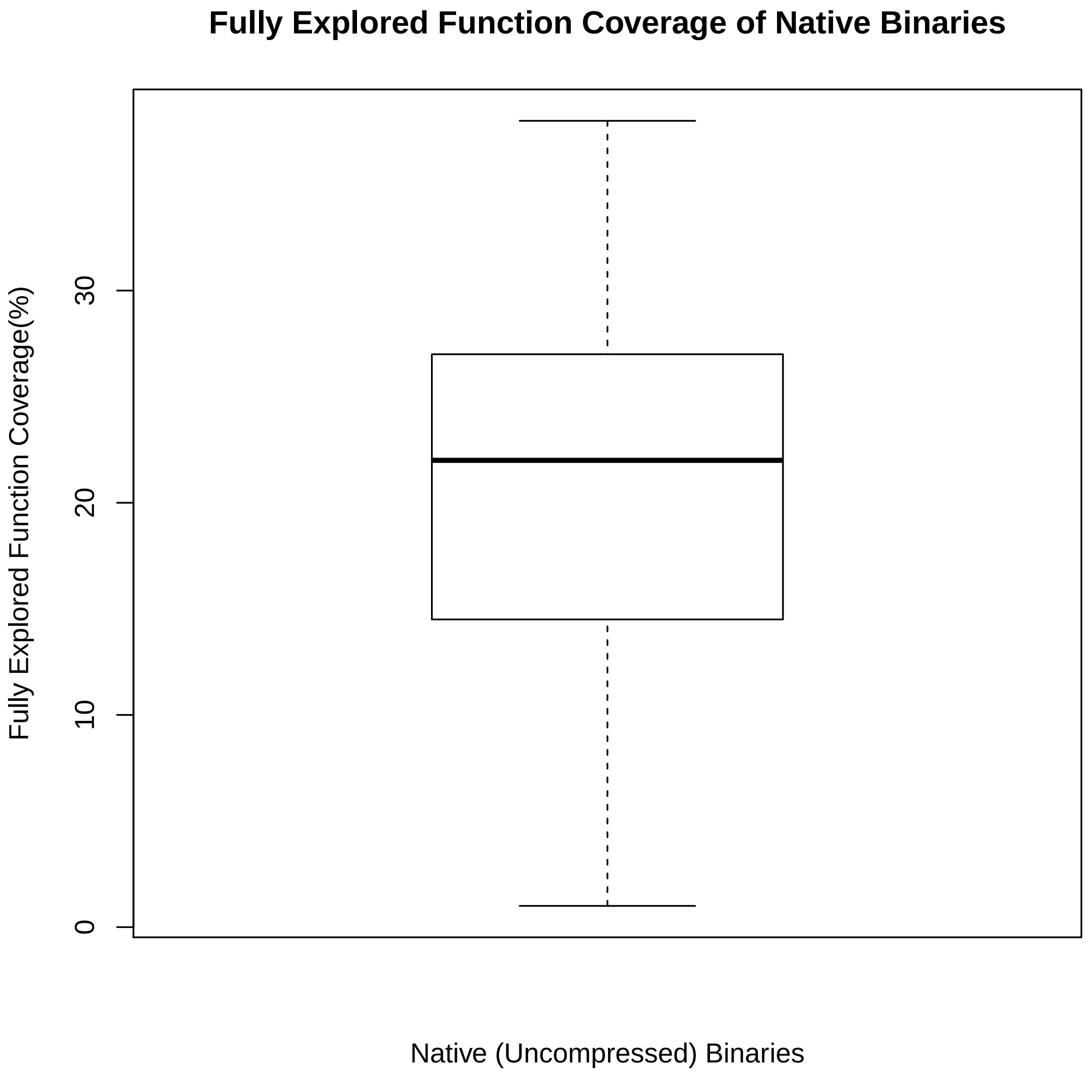


Figure 4.7: Fully explored function coverage of native (uncompressed) binaries.

metric chosen is the number of S2E states/execution paths found during symbolic execution. This metric measures the completeness and accuracy of symbolic execution on obfuscated binaries compared to their native counterparts.

The maximum number of states found, as explained in Section 4.1, is stored in the `run.stats` file. KLEE updates this file whenever a new state is found or an old state has reached the its path termination point and is killed. Since this metric is one of the few that can be measured in obfuscated samples, it is a valuable tool to compare with the original binary. If the number of states in the obfuscated binary is equal to that of those in its native counterpart, then S2E has achieved its maximum potential obfuscated performance.

Since symbolic execution keeps track of more states in an obfuscated binary due to the unpacker code, the number of states found in the obfuscated binaries was anticipated to be lower than the baseline achieved by the non-obfuscated binaries. Unexpectedly, the opposite effect was observed. Section 4.2.1 presents the results from compressing each binary at a mid-range compression level and Section 4.2.2 presents results from binaries at the highest level of compression.

4.2.1 Mid-Level Compression Observations. Figure 4.8 are box plots of the number of states in the native binaries compared to those found at the medium level of compression, `UPX -5`. It appears that the number of states found is higher for the obfuscated samples.

A two-sample t-test determines whether there is a statistically significant difference between the midrange compression level and the uncompressed native baseline. More specifically, Welch's two-sample *t*-test is used since the samples clearly have unequal variances. The one-sided p-value of 1.812 x 10^{-4} is convincing evidence that programs compressed to a mid-range level result in a higher number of states during symbolic execution than those that are uncompressed. The difference in mean states is estimated to be an average of 727.85 higher for mid-level compressed binaries, with a 95% confidence interval of (368.49, 1087.21).

4.2.2 Highest-level Compression Observation. Figure 4.9 are box plots of the number of states found from the native binaries compared to those found at the highest level of compression, `UPX --best`. Again, the obfuscated samples result in more states during symbolic execution than their native versions.

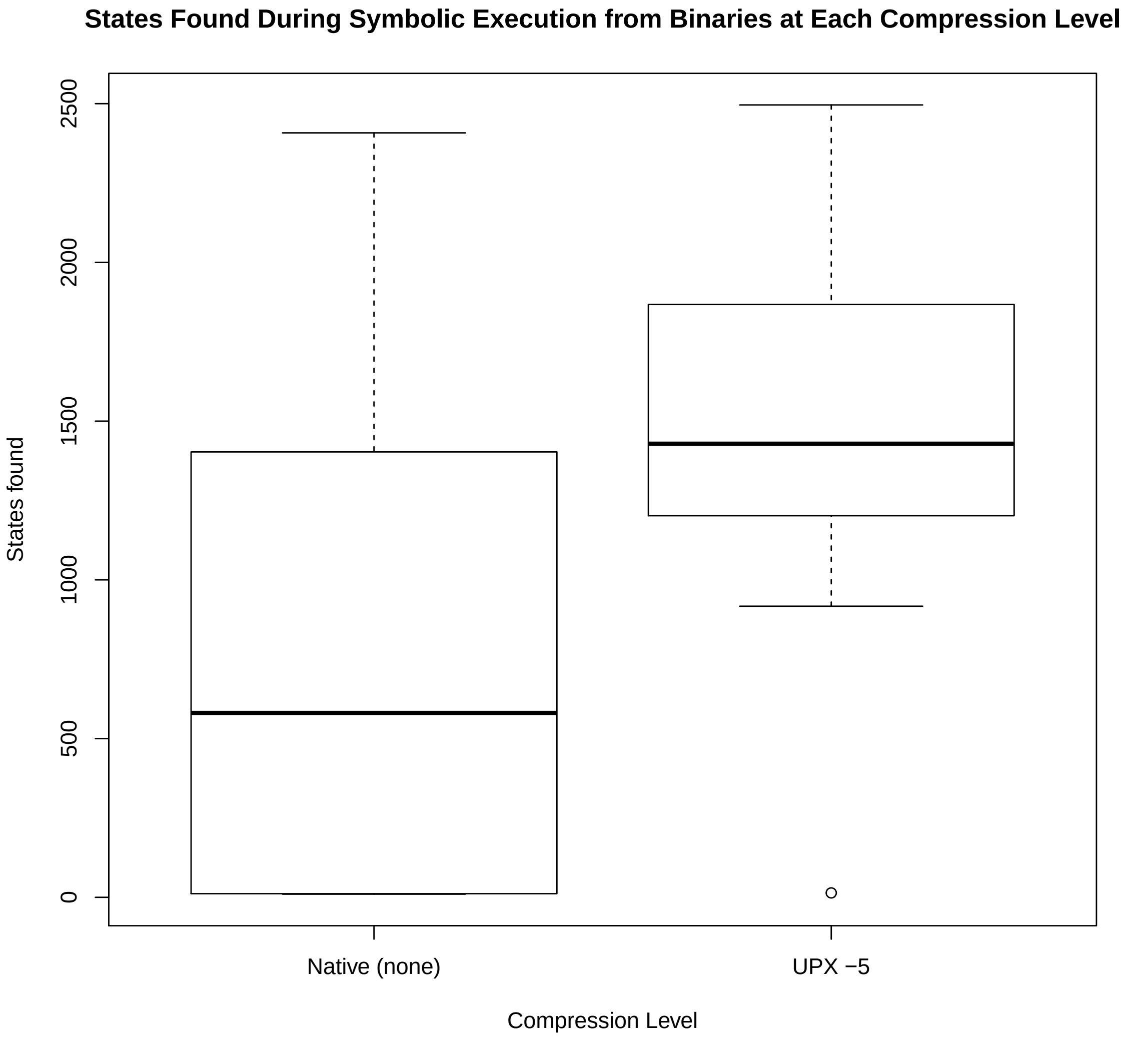


Figure 4.8: States found through symbolic execution from binaries at zero and mid-range compression.

A two-sample t-test is used to determine whether there is a statistically signifcant difference between the number of states found by binaries at the highest compression level and those found by the uncompressed baseline binaries. Welch's two-sample *t*-test is used due to unequal variances. With a one-sided p-value of 1.790 x 10^{-4} there is convincing evidence that binaries packed at UPX's highest level of compression result in a higher number of states during symbolic execution than those that are uncompressed. The

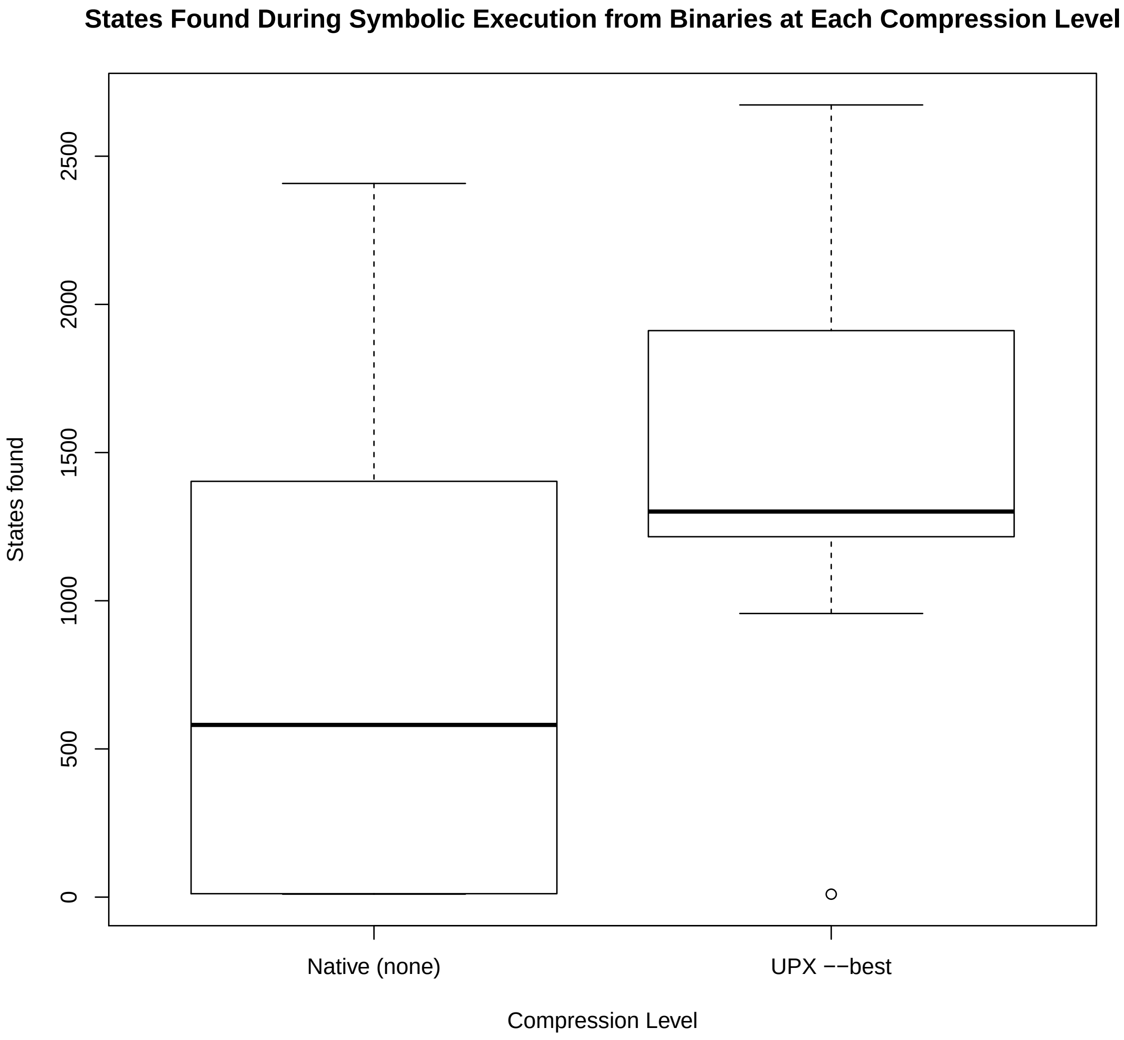


Figure 4.9: States found through symbolic execution from binaries at zero and highest compression.

difference in mean number of states is estimated to be an average of 757.37 higher for the compressed binaries, with a 95% confidence interval of (382.26, 1132.48).

4.2.2.1 Interpretation. The higher number of execution paths found from binaries at both levels of compression compared to their uncompressed counterparts is surprising. There are two explanations for this: 1) the same code is not being symbolically executed in both cases, or 2) the obfuscated code actually lends itself to symbolic execution.

Documentation from the S2E website early in this research suggested its methods for selecting a binary was appropriate for any CLI-based binary. The `init_env` plugin allows arbitrary selection of an executable by first loading the `init_env.so` library with the Linux `LD_PRELOAD` command. This library overrides the initial function call `__libc_start_main` made by every executable and places special S2E-specific methods to enable forking and disable forking around the program code as it is loaded into memory. These methods are encoded as unused processor opcodes and interpreted by the S2E-modified version of QEMU which sends the forking-enabled code through KLEE instead of executing it concretely through QEMU on x86 hardware. This entire mechanism is the basis for S2E's "selective" symbolic execution and it works well for typical binaries that occupy a single, contiguous executable memory space [CW12].

This approach fails for UPX-packed binaries, however, because the header of the compressed binary points immediately to the unpacker code skipping the memory regions which will eventually contain the program's real functionality (cf. Figure 2.3). An interesting case are the uncompressed binaries which resulted in a very low number of paths/states when submitted to S2E for symbolic execution (cf. Table 4.1). If the obfuscated code was actually easier to execute, they should return the same number of state and then terminate. Instead, they returned hundreds or sometimes thousands more paths than the original binary with no compression.

Another interesting phenomenon is the variances for each of the obfuscated portions of the workload are much lower than that of the uncompressed binaries. This difference in variance supports the contention that the same code is not being executed in both cases.

In fact, the actual code submitted to KLEE is the unpacker code, which contains pointers to code in the virtual memory space where each function was assigned to unpack.

Table 4.1: Number of states found at each compression level for all binaries.

Binary Name	None	Mid-level	High
bash	443	2018	1208
busybox	11	1520	1264
chmod	1365	1701	2673
date	1122	1290	1267
dnsdomainname	11	2019	2241
echo	1862	2496	2375
ed	763	969	1111
false	15	14	10
ip	2009	1911	2449
mktemp	10	1864	2055
more	2408	1370	1979
nano	528	1536	1635
netstat	11	1429	1257
pidof	18	1365	1301
ping	13	917	957
ping6	11	1237	990
ps	1536	1935	1672
pwd	12	1106	1211
readlink	1530	1167	1380
rm	942	1588	1127
rmdir	1066	1149	1242
run-parts	11	1777	2483
sed	581	1396	1326
sleep	1441	1285	1299
stty	1300	2008	1844
sync	11	1042	1221
tar	1806	2419	1708

KLEE symbolically executes the unpacker code since this code falls in the first memory space allocated by the executable, but will not symbolically execute any other region in memory including the original binary's code and data sections.

4.3 Recommendations for Improvement

The inability for S2E to detect, choose, and load multiple memory spaces at runtime is the root cause of S2E's tendency to find significantly more paths through the obfuscated binaries than for the original ones. In this section, several improvements are suggested with increasing implementation difficulty. They will not change the fact that S2E is still limited by computing resources and time, but will allow a more thorough symbolic execution of obfuscated binaries themselves (and not just their unpacker code).

4.3.1 Primitive Improvement. The simplest approach to guarantee symbolic execution on the entire obfuscated binary is to remove all memory restrictions for `s2e_enable_forking` and `s2e_disable_forking`. This will effectively introduce the entire memory space (from 0 to (`uint64_t` - 1)) to symbolic execution, allowing the unpacked code from the original binary to enter symbolic execution. However, this will also allow any other library code or kernel-level code required during execution to be run symbolically. Unfortunately, this will also cause the unpacker code to run symbolically as well, exaggerating the path explosion problem. At that point, post-execution becomes the most important problem, as the analyst will need to efficiently differentiate between program code and non-program code.

4.3.2 More Efficient Improvement. The method S2E uses to search the memory address space for the executable to symbolically execute uses the `s2e_get_module_info` function. This function parses a common Linux file at `/proc/self/maps` to find which parts of memory belong to a process. Displaying this file during runtime reveals many spaces are allocated by the unpacker code, but are left unnamed. An addition could be made to `s2e_get_module_info` to also parse memory address spaces that have no name. If an execution path enters any of these regions, symbolic execution can be performed. If execution does not enter one or more of these regions, there is no loss in performance. As a performance bonus, if a binary is packed (trivial to determine if more than one memory region is marked executable), execution can be explicitly denied on the memory space originally allocated to the packed executable as that space should contain only unpacker code which should be concretely executed.

4.3.3 Advanced Improvement. A more advanced method for improving S2E is to override the `mprotect` method. This is a common Linux function that controls the privileges of memory blocks. When new memory is allocated for a process or when the read/write/execute privileges are changed, `mprotect` is called to do this. Therefore, it could be overridden to add any new executable memory regions to the current program being symbolically executed whenever these privileges are changed. The current `init_env.so` library already overrides `__libc_start_main`, and so the addition of another override should work as well.

4.4 Summary

The performance of S2E on native, uncompressed binaries demonstrates the current practical shortcomings of symbolic execution compared to static disassembly with respect to code coverage. The use of execution states (i.e., paths explored) provides a valuable baseline metric with which to compare binary samples that IDA Pro cannot correctly disassemble, and on which S2E cannot provide coverage statistics.

S2E's behavior over binaries at medium and high obfuscation levels identified deficiencies in its implementation to dynamically select appropriate memory bounds for symbolic execution. In response to these findings, three improvements are presented to include all relevant code sections in obfuscated samples, thereby increasing code coverage and the capability that S2E has in reasoning about packed executables.

5 Conclusions

This chapter presents the significance of this research in Section 5.1. Ideas for future work are described in Section 5.2, and final thoughts are given in Section 5.3.

5.1 Significance of Research

This research used the Selective Symbolic Execution (S2E) engine to measure the effectiveness of symbolic execution as a binary analysis platform. It established coverage baselines with respect to basic blocks, touched functions, and fully explored functions on a sample set of real-world Linux binaries setting expectations for future utility of symbolic execution.

This research also tested the applicability of S2E for obfuscated binaries at several levels and determined S2E does not currently support symbolic execution on programs split into multiple executable memory regions at runtime (i.e., most packed code). Instead, S2E simply uses the first memory region available to the current process, which in early testing of non-obfuscated binaries resulted in the correct region.

Given these results, three recommendations for improvement to S2E are made. All three recommendations could solve the obfuscation problem, albeit at different levels of eloquence and efficiency. The first method submits all possible memory space for symbolic execution, but will likely not be selective enough to offer much useful data. The second method is much more eloquent, but will need to be fine-tuned for the behavior of different packing tools depending on how the new memory regions allocated are named. The final method will allow universal recognition of runtime unpacker code that needs to change the read/write/execute privileges of memory regions, and should be most effective.

5.2 Future Work

There is clearly much work to be done before symbolic execution can be used with the same frequency as other binary analysis methods. Nevertheless, several potential problems stand out and should be fixed first.

5.2.1 Code Priority. The theory behind S2E was to prioritize certain sections of code before others with the observation that only portions of the code are considered "interesting." This is important, because many portions of a piece of malware may be in reused libraries that have already been analyzed; time and effort spent to analyze these again would be a waste. Instead of attempting to symbolically execute entire binaries as in this research, it would be beneficial to design a method to categorize new sections of code and prioritize/triage them appropriately with S2E.

5.2.2 Baselines for Obfuscated Code. Even after accomplishing the recommended improvements with respect to obfuscated binaries, quantifying code coverage is still an impossible task with S2E's current coverage methods. The current implementation of S2E's `coverage` tool depends on viable `.bblist` output from IDA Pro and the S2E-provided `extractBasicBlocks.py` script. However, IDA cannot effectively disassemble obfuscated binaries since they change at runtime. Providing a method to verify code coverage on obfuscated (and uncompressed) binaries without IDA Pro is desperately needed to make an informed decision on the percentage of an obfuscated binary that can actually be reasoned about.

5.2.3 Handling and Testing of Obfuscated Code. An interesting future research project could implement and test all three recommendations given at the end of Chapter 4. Using these recommendations as metrics for obfuscated binary analysis, one could determine which offers the best speed, accuracy, and completeness of symbolic execution compared to a given baseline. The baseline could be determined through symbolic execution of uncompressed binaries, as in this research, or through other methods that only used obfuscated binaries as a baseline.

5.3 Conclusion

Symbolic execution, at least in the implementation offered by S2E, is not yet ready for the same rigors of binary analysis that the combination of program debuggers and static disassemblers currently handle. The practical, real-world limitations of symbolic execution manifest themselves on small executables, all of them less than one megabyte in size. As a result, symbolic execution should only be used for unknown executables that resist traditional debugging/disassembly techniques. However, many of the shortcomings experienced by S2E for symbolically executing obfuscated binaries occupying multiple memory spaces can be overcome and would provide a single, automated, and unified platform for all types of binary analysis. A higher theoretical potential for path discovery than either of the traditional methods of reverse engineering and the ability for execution paths to be automatically explored and made suitable for mathematical analysis, as well as automatically creating and returning relevant test cases is a tantalizing vision of future malware analysis.

Appendix A: Binaries Used in Experiment, with Descriptions

A.1 bash

The standard GNU Bourne-Again SHell. A sh-compatible command language interpreter that executes commands read form the standard input or from a file. Bash also incorporates useful features from the Korn and C shells (ksh and csh). Bash is intended to be a conformant implementation of the Shell and Utilities portion of the IEEE POSIX specification (IEEE Standard 1003.1). Bash can be configured to be POSIX-conformant by default.

A.2 busybox

The Swiss Army Knife of Embedded Linux. Busybox combines tiny versions of many common UNIX utilities into a single small executable. It combines minimalist replacements for most of the utilities you usually find in GNU coreutils, util-linux, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts.

A.3 chmod

Changes the file mode bits of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

A.4 date

Print or set the system date and time.

A.5 dnsdomainname

Print the domain part of the FQDN (Fully Qualified Domain Name).

A.6 echo

Display a line of text.

A.7 ed

Ed is a line-oriented text editor. It is used to create, display, modify and otherwise manipulate text files. red is a restricted ed: it can only edit files in the current directory and cannot execute shell commands. If invoked with a file argument, then a copy of file is read into the editor's buffer. Changes are made to this copy and not directly to file itself. Upon quitting ed, any changes not explicitly saved with a 'w' command are lost.

A.8 false

Exit with a status code indicating failure.

A.9 ip

Show/manipulate routing, devices, policy routing and tunnels.

A.10 mktemp

Create a temporary file or directory, safely, and print its name.

A.11 more

More is a filter for paging through text one screenful at a time. This version is especially primitive. Users should realize that less(1) provides more(1) emulation and extensive enhancements.

A.12 nano

Nano is a small, free and friendly editor which aims to replace Pico, the default editor included in the non-free Pine package. Rather than just copying Pico's look and feel, nano also implements some missing (or disabled by default) features in Pico, such as ”search and replace” and ”go to line and column number”.

A.13 netstat

Netstat prints information about the Linux networking subsystem.

A.14 pidof

Pidof finds the process id's (pids) of the named programs. It prints those id's on the standard output.

A.15 ping

Ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (”pings”) have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of ”pad” bytes used to fill out the packet.

A.16 ping6

Ping6 uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (”pings”) have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of ”pad” bytes used to fill out the packet. Designed for Internet Protocol version 6.

A.17 ps

Ps displays information about a selection of the active processes.

A.18 pwd

Print the full filename of the current working directory.

A.19 readlink

Readlink() places the contents of the symbolic link path in the buffer buf, which has size bufsiz.

A.20 rm

Remove files or directories.

A.21 rmdir

Rmdir() deletes a directory, which must be empty.

A.22 run-parts

Runs all scripts or programs in a directory.

A.23 sed

Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as ed), sed works by making only one pass over the input(s), and is consequently more efficient. But it is sed's ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

A.24 sleep

Sleep() makes the calling thread sleep until seconds seconds have elapsed or a signal arrives which is not ignored.

A.25 stty

Change and print terminal line settings

A.26 sync

Sync writes any data buffered in memory out to disk.

A.27 tar

GNU 'tar' saves many files together into a single tape or disk archive, and can restore individual files from the archive.

Appendix B: Source Code of Scripts Written or Used

B.1 ListTB2.py

Description: Written in Python 2.x, compares the execution trace files found from the included `tbtrace` tool from S2E, and builds a master file containing all the unique PC addresses found.

```
’’’
Created on Jan 25, 2012

@author: Jon Miller

Given a number of paths from S2E’s output (0.txt, 1.txt, etc. up to 10k),
this will list all the different Translation Blocks that were executed.

Output is a temp.txt file which contains the different lines from all files,
and a final.txt file which contains only PC addresses of the TBs.
’’’

t = open(’temp.txt’,’w’)
t.close()

# compare files
def compareFile(inFile):
    for inLine in inFile.readlines():
outputFile = open(’temp.txt’,’r+’)
        #scan and compare current file with inLine
        if not (inLine in outputFile.readlines()):
            outputFile.write(inLine)
     outputFile.close()

# access files
def accessFiles():
    for x in range(0,10000):
        try:
            inputFile = open(str(x) + ’.txt’, ’r’)
            compareFile(inputFile)
            inputFile.close()
            print("Analyzed path " + str(x) + ’\r’),
        except:
            break
    print   # clears the next line so that the progress isn’t
    # overwritten by the next terminal prompt

# remove error messages, forks, etc leaving only PC addresses of TBs
def cleanup():
    f = open(’final.txt’,’w’)
    for line in open(’temp.txt’,’r’):
#line starts with 0x, must be an address
if((line[0] == ’0’) and (line[1] == ’x’)):
    f.write(line)
    f.close()
```

```
def main():
    accessFiles()
    cleanup()


if __name__ == '__main__':
    main()
```

B.2 extractBasicBlocks.py

Description: Written in Python 2.x, and using IDA Pro's disassembly, outputs to file the beginning PC address, ending PC address, and name for each function found.

```
import idaapi

import idautils

import idc


#Flowchart does not consider function calls as basic block boundaries.

#This function takes a range of addresses and splits additional basic blocks.

def cls_split_block(fp, startEA, endEA):

  curName = GetFunctionName(startEA);

  dem = idc.Demangle(curName, idc.GetLongPrm(INF_SHORT_DN));

  if dem != None:

    curName = dem;


  first=startEA

  h = idautils.Heads(startEA, endEA)

  for i in h:

    mnem = idc.GetMnem(i)

    if mnem == "call" and i != endEA:

      print >>fp, "%#010x %#010x %s" % (first,

        idc.NextHead(i, endEA+1)-1,

        curName)

      first=idc.NextHead(i, endEA+1)
  if first < endEA:
    print >>fp, "%#010x %#010x %s" % (first, endEA-1, curName)

# ----------------------------------------------------------------------
# Using the class
def cls_main(fp, func, p=True):
  global f
  f = idaapi.FlowChart(idaapi.get_func(func))
  for block in f:
  cls_split_block(fp, block.startEA, block.endEA)
  #if p: print >>fp, "%#10x %#10x" % (block.startEA, block.endEA)
    #for succ_block in block.succs():
    #    if p: print "  %x - %x [%d]:" % (succ_block.startEA,
    #      succ_block.endEA, succ_block.id)
    #for pred_block in block.preds():
    #    if p: print "  %x - %x [%d]:" % (pred_block.startEA,
    #      pred_block.endEA, pred_block.id)

def extract_bbs():
  filename = idc.AskFile(1, "*.*", "Save list of basic blocks")
    exit = False
    if not filename:
      basename = idc.GetInputFile()
      filename = basename + ".bblist"
      idc.GenerateFile(idc.OFILE_ASM, basename + ".asm", 0, idc.BADADDR, 0)
      idc.GenerateFile(idc.OFILE_LST, basename + ".lst", 0, idc.BADADDR, 0)
      exit = True
  fp = open(filename,’w’)
  funcs = idautils.Functions()
  for f in funcs:
    cls_main(fp, f)
      if exit:
        idc.Exit(0)


q = None
f = None
idc.Wait()
extract_bbs()
```

B.3 packBinaries.py

Description: Written in Python 2.x, automates binary packing.

```
’’’
Created on Jan 28, 2012

@author: Jon Miller

Used to automate the process of packing all the binaries extracted from
the guest Linux machine to the ’host’ Linux machine.

Uses UPX version 3.08. Given the set of executables, it will compress each
of them and place them in the correct upx_# folder, where # is the
compression level used (1,2,3,4,5,6,7,8,9,best).
’’’
import subprocess

# set of all executables
bins = [’bash’,’busybox’,’cat’,’chgrp’,’chmod’,’chown’,’cp’,’cpio’,’date’,
’dd’,’df’,’dir’,’dmesg’,’dnsdomainname’,’echo’,’ed’,’egrep’,’false’,
        ’fgconsole’,’fgrep’,’grep’,’gzip’,’hostname’,’ip’,’kill’,’ln’,
        ’loadkeys’,’login’,’ls’,’lsmod’,’mkdir’,’mknod’,’mktemp’,’more’,
’mount’,’mountpoint’,’mt’,’mt-gnu’,’mv’,’nano’,’nc’,’nc.traditional’,
’netcat’,’netstat’,’pidof’,’ping’,’ping6’,’ps’,’pwd’,’rbash’,
’readlink’,’rm’,’rmdir’,’rnano’,’run-parts’,’sed’,’sh’,’sleep’,
’stty’,’su’,’sync’,’tailf’,’tar’,’tempfile’,’touch’,’true’,’umount’,
’uname’,’vdir’]
```

```
upxdir = '/home/s2e/Desktop/upx'
bindir = '/home/s2e/Desktop/guestbins'

def upx():
    for eachbin in bins:
        #goes from compression rates 1 to 9
        for x in range(1,10):
            subprocess.call([upxdir+'/upx',         #use upx
                '-'+str(x),                         #compression rate
                '-o',                               #output to...
                bindir+'/upx_'+str(x)+'/'+eachbin+'_'+str(x), #output name
                bindir+'/bin/'+eachbin])             #source binary)
        #best compression
        subprocess.call([upxdir+'/upx',             #use upx
            '--best',                               #compression rate
            '-o',                                   #output to...
            bindir+'/upx_best/'+eachbin+'_best',    #output name
            bindir+'/bin/'+eachbin])                #source binary

def main():
    upx()

if __name__ == '__main__':
    main()
```

B.4 runTBTrace.py

Description: Written in Python 2.x, automates creation of execution traces.

```
’’’
Created on Feb 10, 2012

@author: Jon Miller

Used to automate the process of creating /traces folder within each
result, and running tbtrace on that.

Can be executed from any location, but leave it on Desktop for later
reference.
’’’
import subprocess

# set of all executables used
bins = [’bash’,’busybox’,’chmod’,’date’,’dnsdomainname’,’echo’,’ed’,
’false’,’ip’,’mktemp’,’more’,’nano’,’netstat’,’pidof’,’ping’,
’ping6’,’ps’,’pwd’,’readlink’,’rm’,’rmdir’,’run-parts’,’sed’,
’sleep’,’stty’,’sync’,’tar’]

resultsdir  = ’/home/s2e/Desktop/s2eResults/’
tbtrace     = ’/home/s2e/s2e/build/tools/Release/bin/tbtrace’
outputdir   = ’-outputdir=’
trace       = ’-trace=’
```

```
def traceit():
    for eachbin in bins:

        #for native binaries
#first mkdir /traces/
        subprocess.call([’mkdir’,
        resultsdir+eachbin+’/traces/’])
#then run tbtrace
        subprocess.call([tbtrace,
            outputdir+resultsdir+eachbin+’/traces/’,
            trace+resultsdir+eachbin+’/ExecutionTracer.dat’])

        #for UPX_5 binaries
#first mkdir /traces/
        subprocess.call([’mkdir’,
        resultsdir+eachbin+’_5’+’/traces/’])
#then run tbtrace
        subprocess.call([tbtrace,
            outputdir+resultsdir+eachbin+’_5’+’/traces/’,
            trace+resultsdir+eachbin+’_5’+’/ExecutionTracer.dat’])

        #for UPX_best binaries
#first mkdir /traces/
        subprocess.call([’mkdir’,
        resultsdir+eachbin+’_best’+’/traces/’])
#then run tbtrace
```

```
        subprocess.call([tbtrace,
            outputdir+resultsdir+eachbin+’_best’+’/traces/’,
            trace+resultsdir+eachbin+’_best’+’/ExecutionTracer.dat’])

        #no status report/print statement needed, tbtrace already does that


def main():
    traceit()

if __name__ == ’__main__’:
    main()
```

B.5 config.lua

Description: Written in Lua, this script sets the configuration options for S2E and KLEE. It is required by S2E and KLEE to run properly.

```
-- Configuration file for enabling Plugins within S2E and
-- manipulating KLEE execution.
-- This was distributed along with all the other S2E tools, and
-- S2E needs it to run.
-- It should be kept in /home/s2e/s2e/, the same directory as
-- launch-x86, launch-x86-orig, and the qcow2 Debian disk image.

s2e = {
  kleeArgs = {
  -- Run each state for at least 5 sec before
  -- switching to the other:
    "--use-batching-search=true",
    "--batch-time=5.0",
    "--state-shared-memory=true"
  }
}



plugins = {
  -- Enable S2E custom opcodes
  "BaseInstructions",

  -- [Not available in this S2E installation]
  -- Enable SimpleSelect for use with init_env
  -- "SimpleSelect",

  -- Track when the guest loads programs
  "RawMonitor",

  -- Detect when execution enters the
  -- program of interest
  "ModuleExecutionDetector",

  -- Restrict symbolic execution to
  -- the programs of interest
  "CodeSelector",

  -- Required by TBTracer
  "ExecutionTracer",

  -- To give basic blocks executed
  "TranslationBlockTracer",

  -- Need so we know which module the traced
  -- program counters belong to.
  "ModuleTracer"
}

pluginsConfig = {}
```

B.6 launch-x86-orig

Description: Written in shell scripting code, launches the original QEMU system emulator and boots the named disk image. Loads the previous snapshot of this disk image if there is one.

```
#!/bin/sh
S2E=/home/s2e/s2e/build/qemu-release/i386-softmmu/qemu
$S2E -hda debian_lenny_i386_standard.qcow2 -loadvm 1
```

B.7 launch-x86

Description: Written in shell scripting code, launches the S2E-modified version of the QEMU system emulator and boots the named disk image. Note that the config.lua file from Appendix B.5 is used when launching S2E. Also loads the previous snapshot of this disk image if there is one.

```
#!/bin/sh
S2E=/home/s2e/s2e/build/qemu-release/i386-s2e-softmmu/qemu
$S2E -hda debian_lenny_i386_standard.qcow2 -loadvm 1 \
-s2e-config-file config.lua -s2e-verbose
```

B.8 send-mail

Description: Written in shell scripting code, sends the experimenter an email. Typical usage was immediately after the previous script in Appendix B.7 (launched together), so the command would be: `sh launch-x86; sh send-mail`. Also, the "`VM X`" would be replaced by the VM number that it happened to be running on. Before using on a clean Ubuntu install, the exim4-light-daemon (or equivalent mail daemon) must be installed. Instructions for exim4 configuration were followed from [http://www.manu-j.com/blog/wordpress-exim4-ubuntu-gmail-smtp/75/].

```
#!/bin/sh
date | mail -s "[Run is done on VM X.]" [personal email redacted]@gmail.com
```

Appendix C: Extraneous Graphs and Run Data

C.1 Basic Block Scatterplots

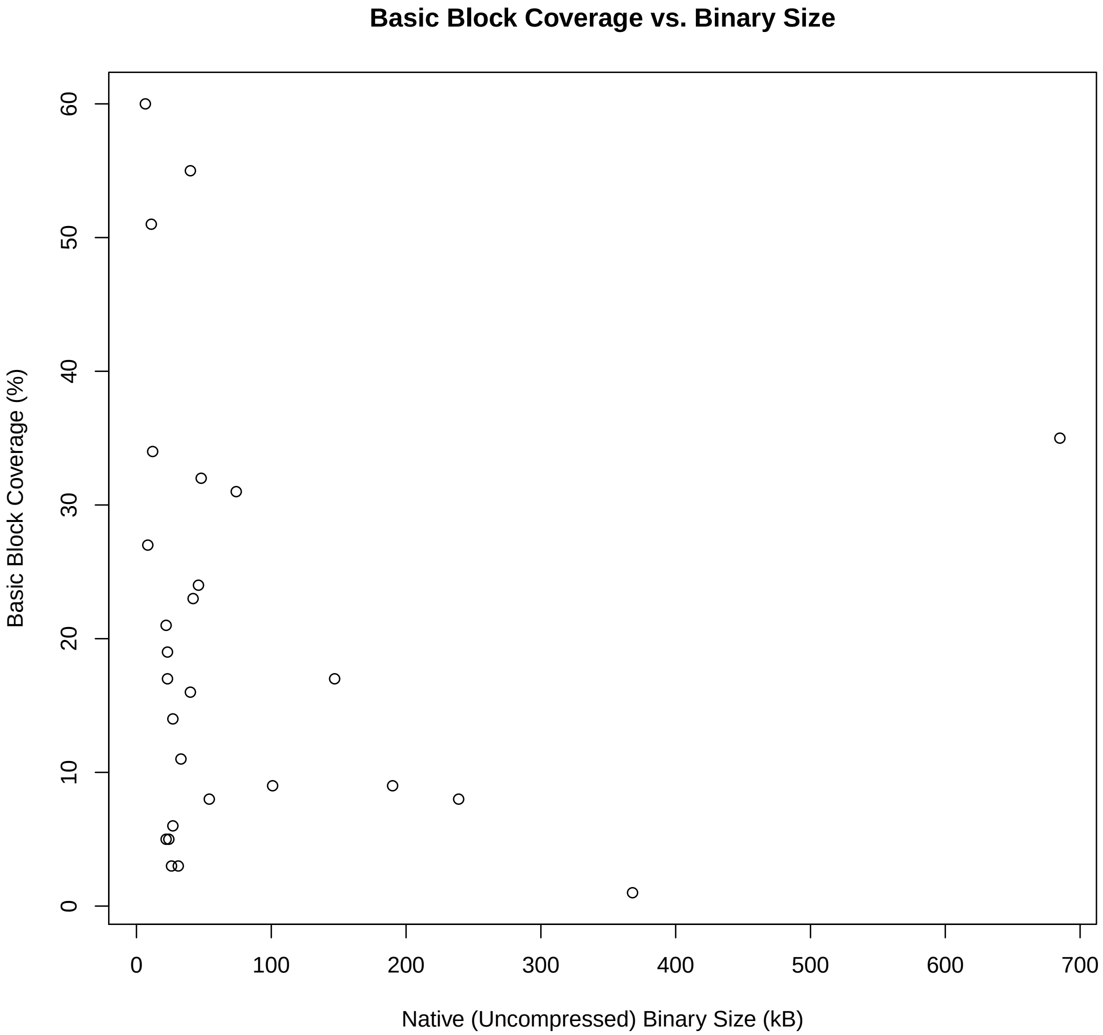


Figure C.1: Scatterplot of Basic Block Coverage vs. Binary Size.

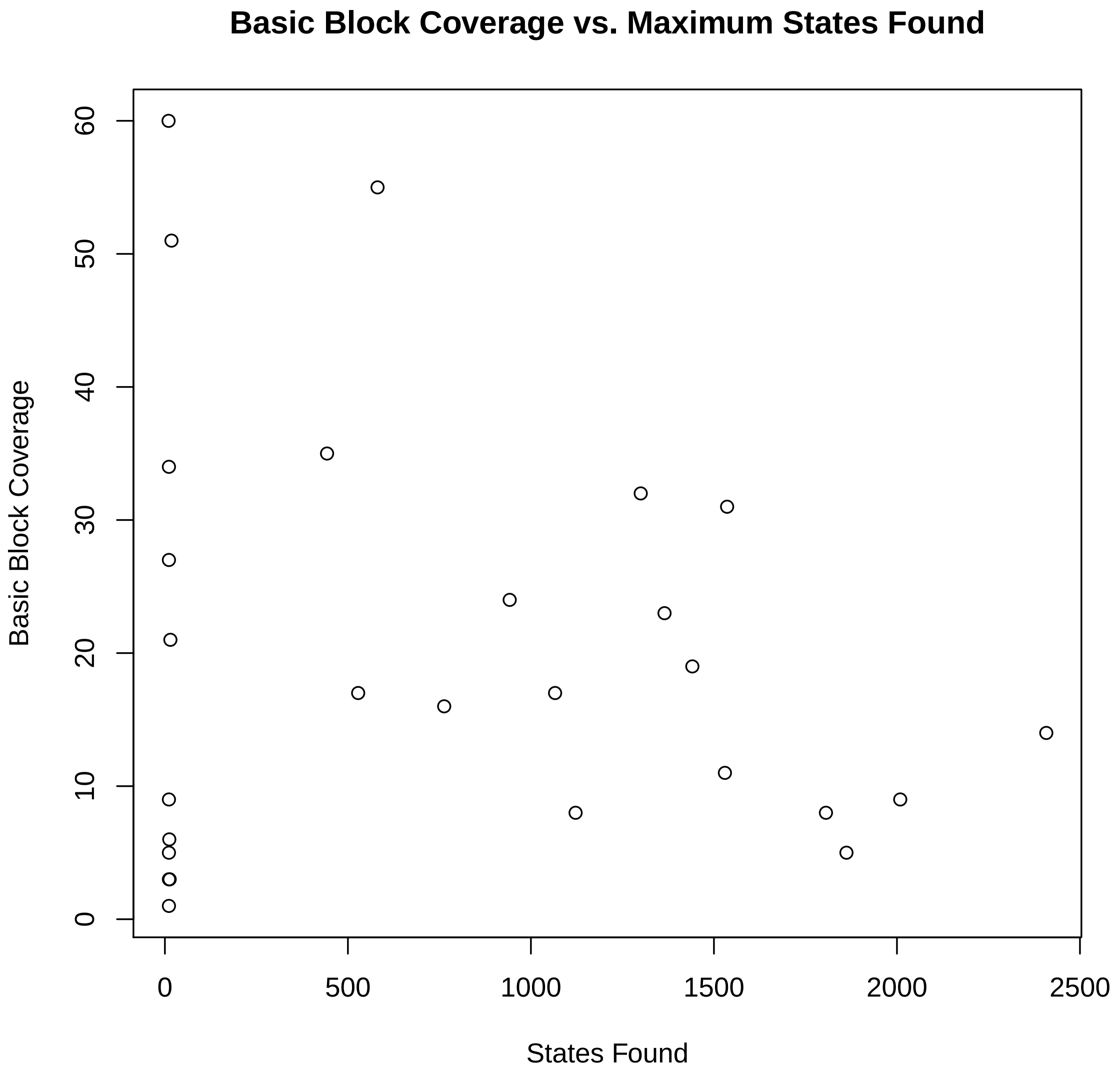


Figure C.2: Scatterplot of Basic Block Coverage vs. Max States Found.

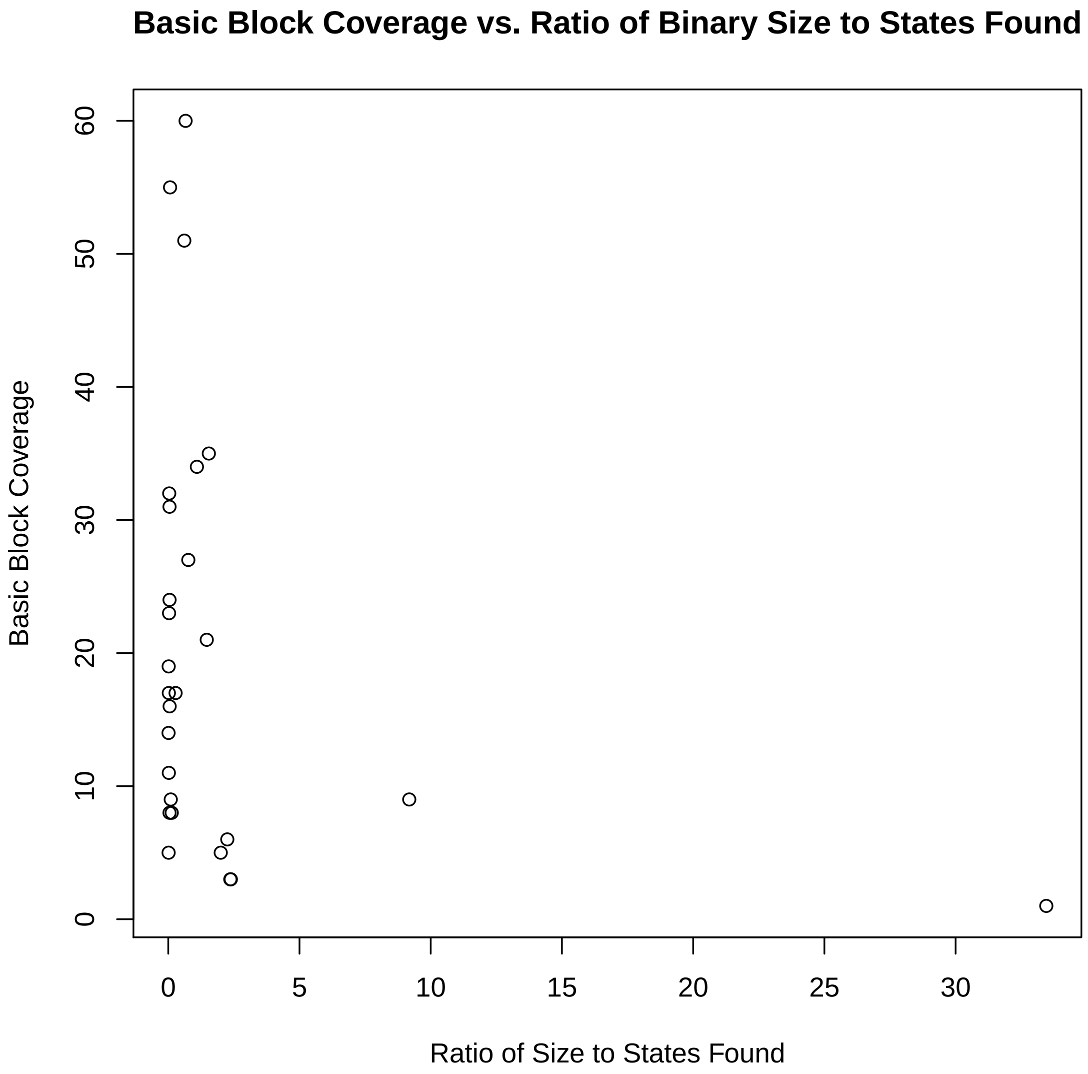


Figure C.3: Scatterplot of Basic Block Coverage vs. Ratio of Binary Size to Max States Found.

C.2 Touched Function Block Scatterplots

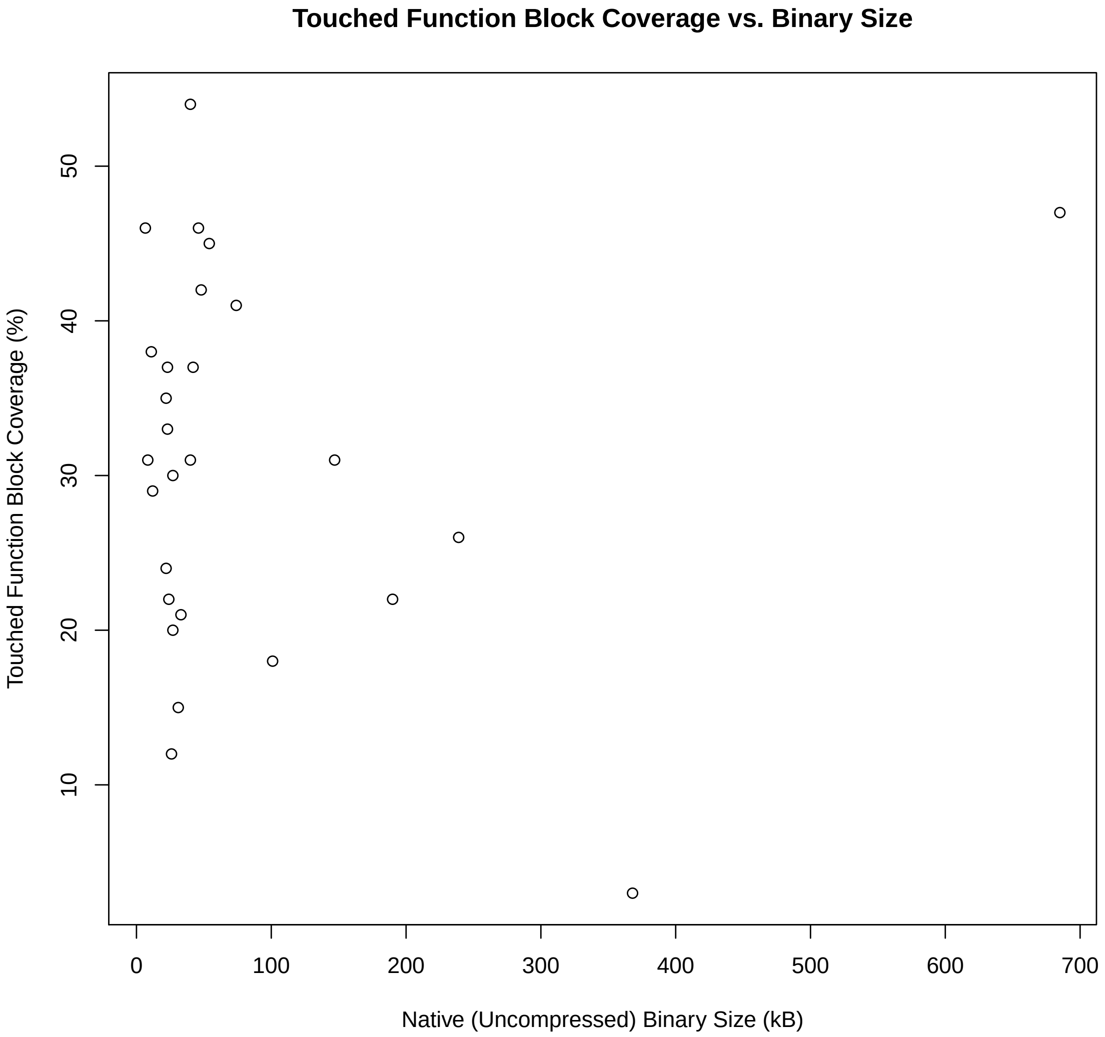


Figure C.4: Scatterplot of Touched Function Block Coverage vs. Binary Size.

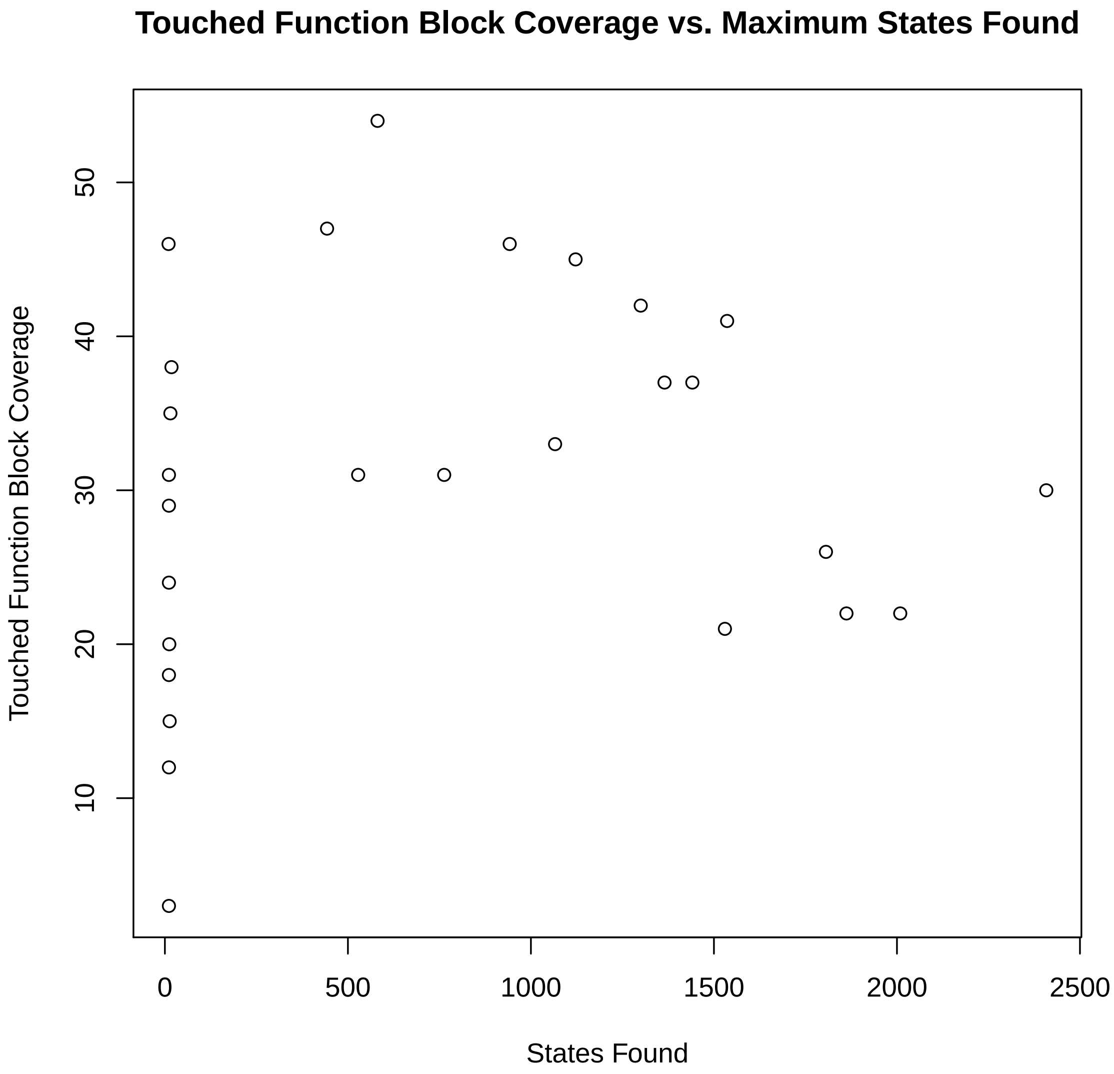


Figure C.5: Scatterplot of Touched Function Block Coverage vs. Max States Found.

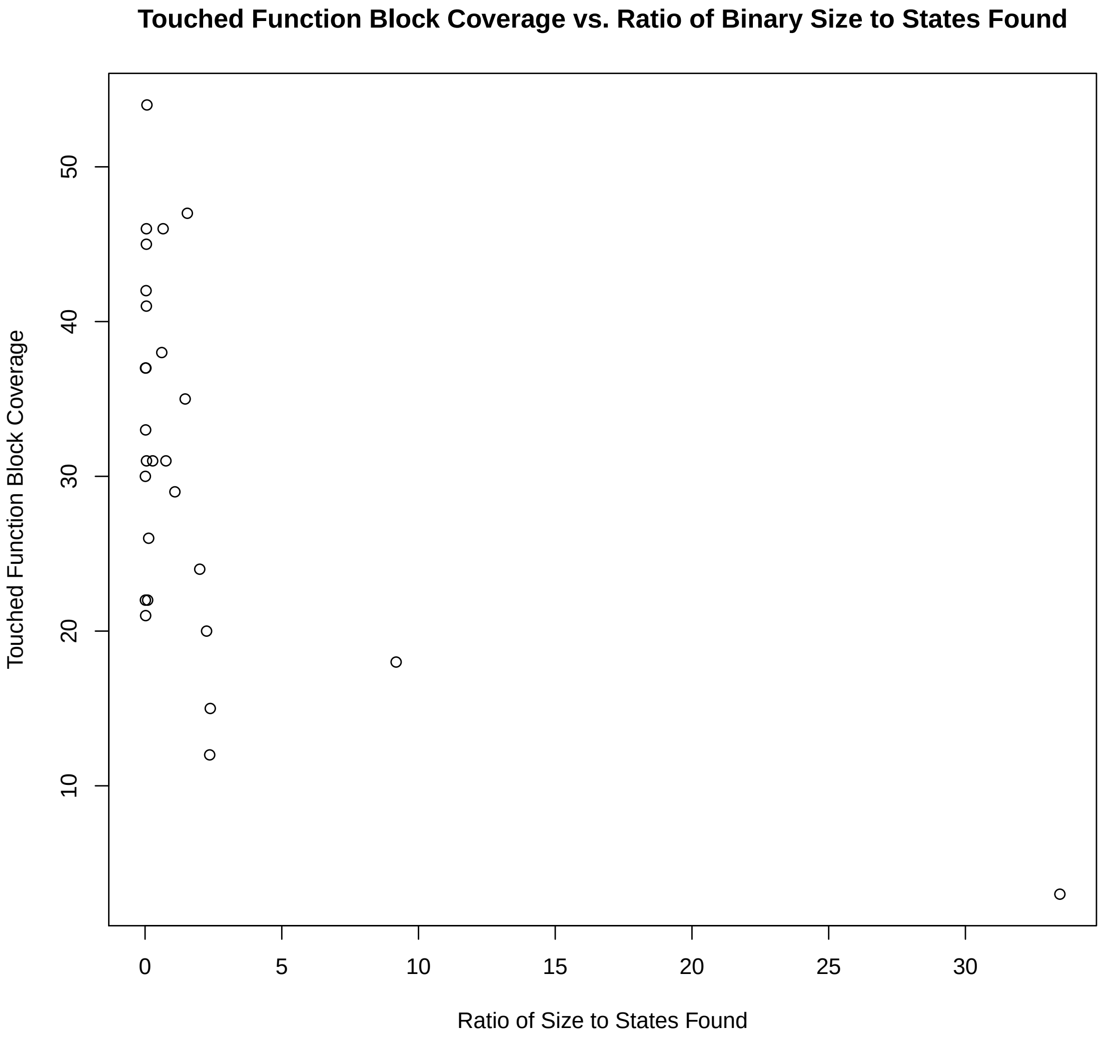


Figure C.6: Scatterplot of Touched Function Block Coverage vs. Ratio of Binary Size to Max States Found.

C.3 Fully Explored Function Block Scatterplots

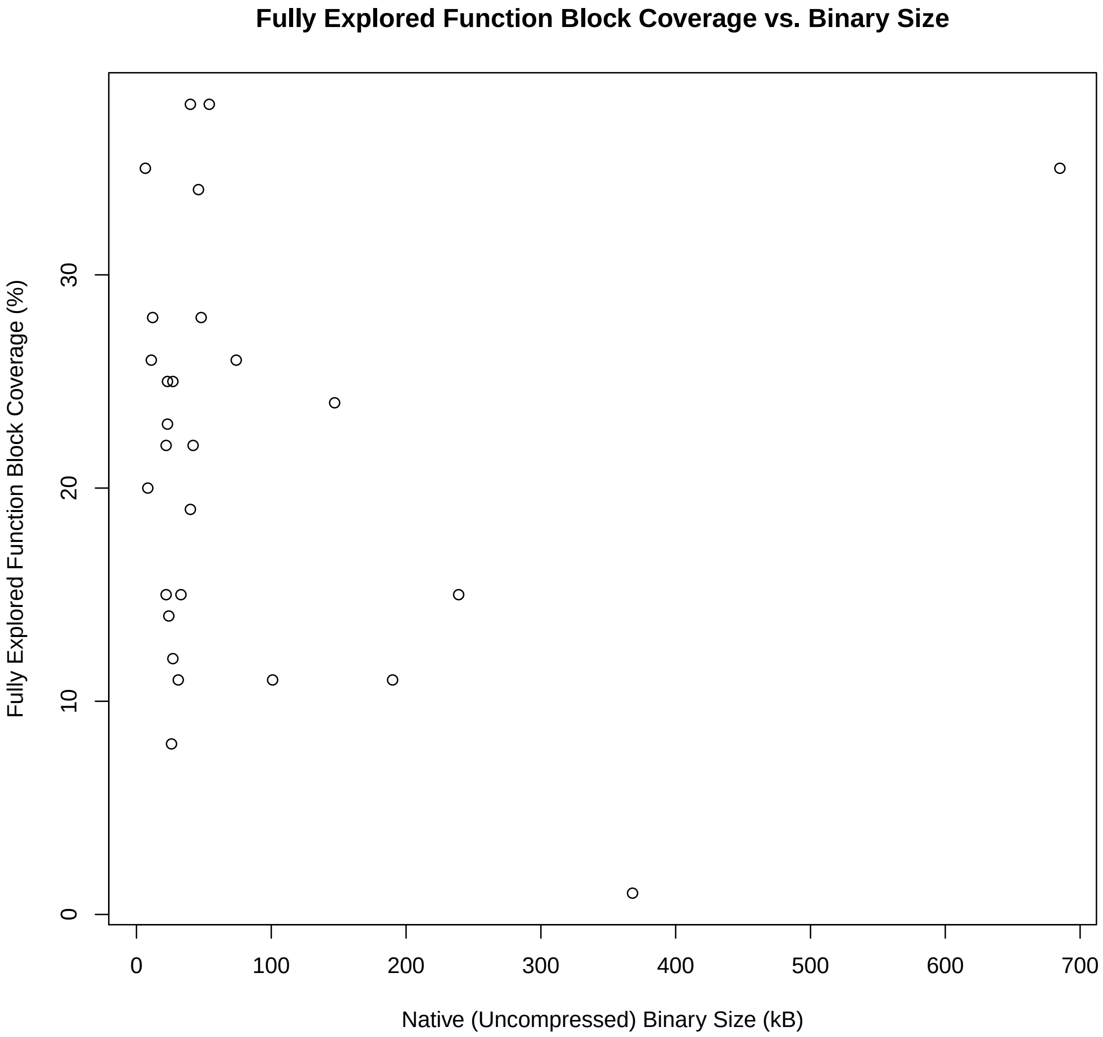


Figure C.7: Scatterplot of Fully Explored Function Block Coverage vs. Binary Size.

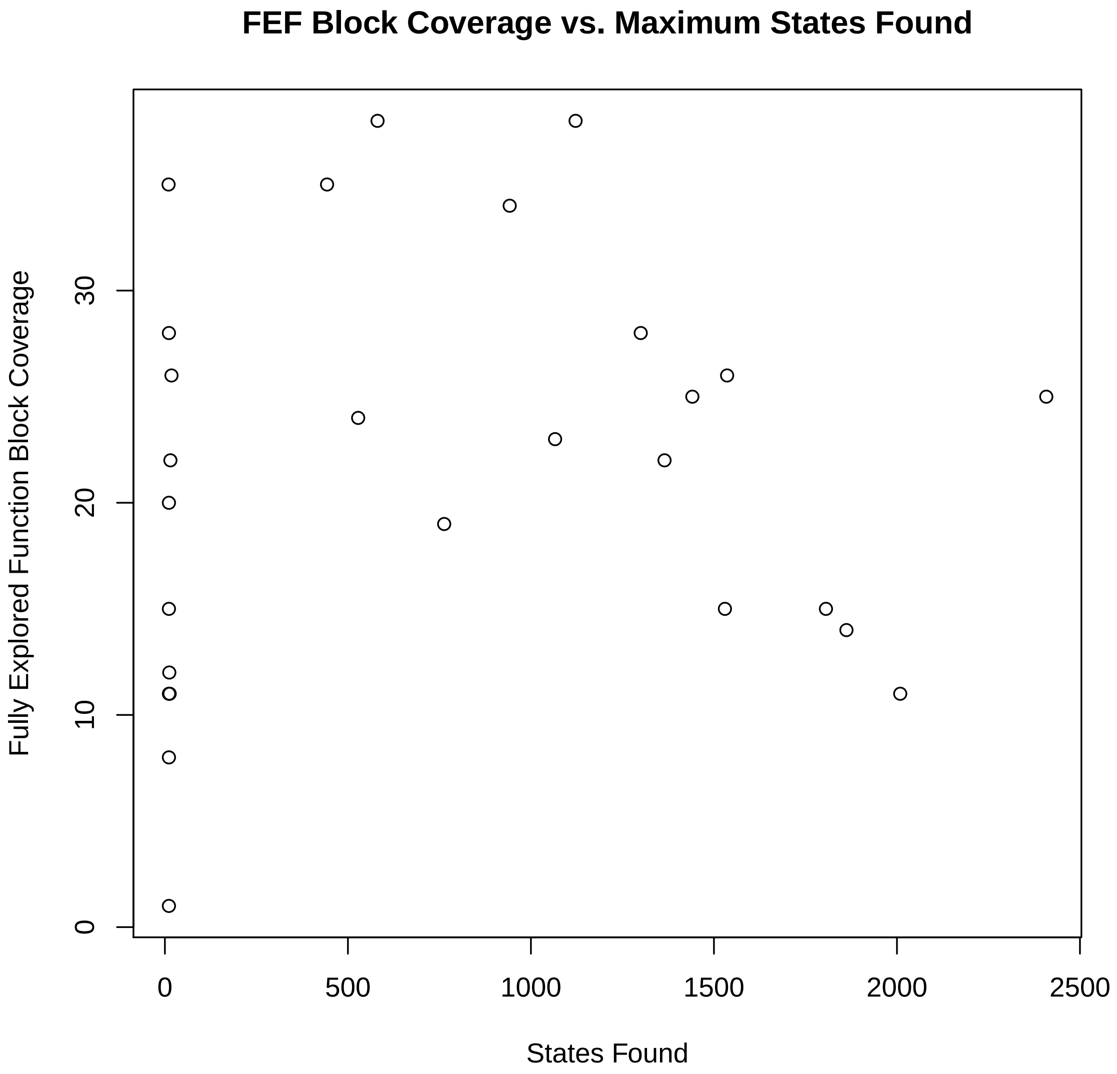


Figure C.8: Scatterplot of Fully Explored Function Block Coverage vs. Max States Found.

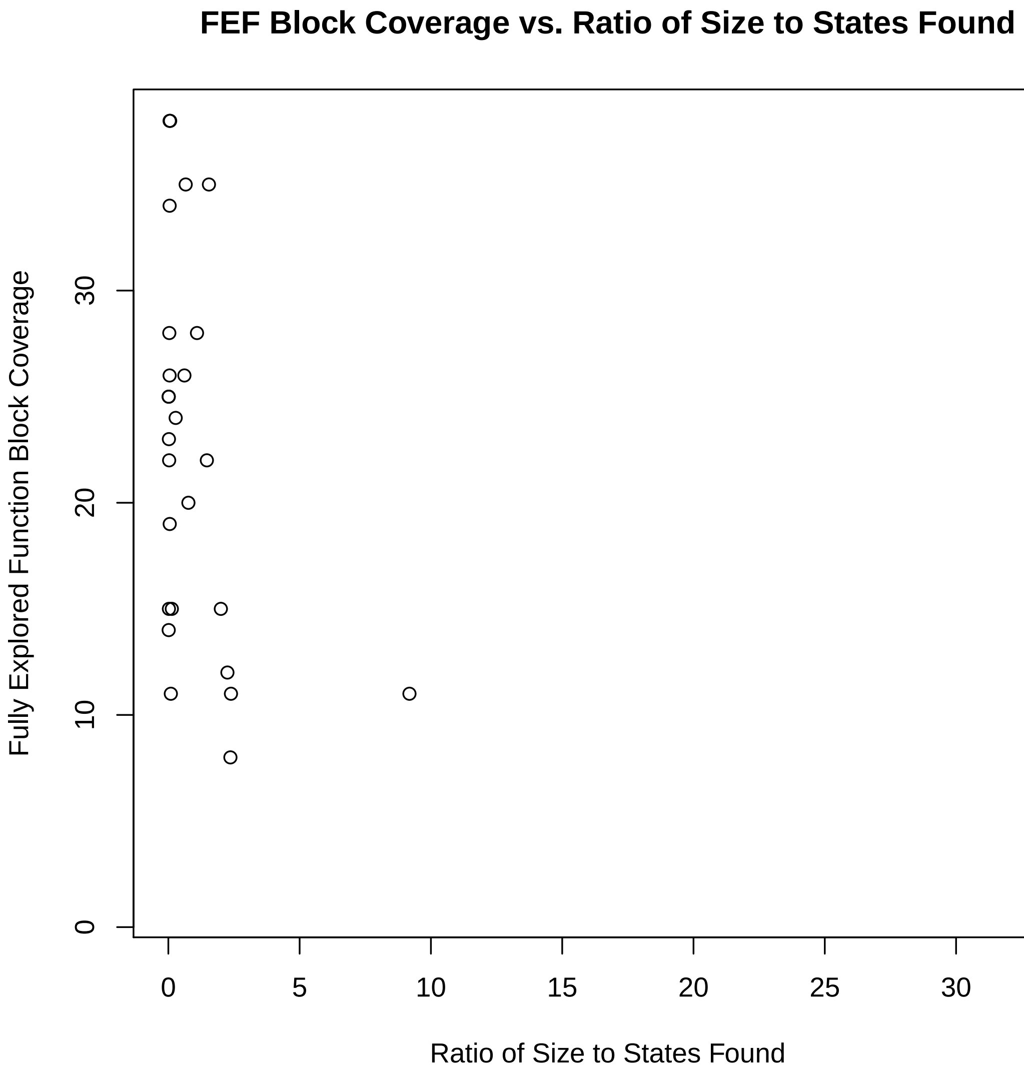


Figure C.9: Scatterplot of Fully Explored Function Block Coverage vs. Ratio of Binary Size to Max States Found.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[Bel11] Fabrice Bellard. *QEMU: Open Source Processor Emulator*. http://www.qemu.org, 2011.

[BR04] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, pages 5–23, 2004.

[CC10] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, Ecole Polytechnique Federale de Lausanne, Switzerland, 2010.

[CCKK12] Vitaly Chipounov, George Candea, Andreas Kirchner, and Volodymyr Kuznetsov. *S2E Source Code Repository*. Ecole Polytechnique Federale de Lausanne (EPFL), Feb 2012. https://s2e.epfl.ch/projects/s2e/repository.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'08 Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[Cif94] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, March 2011.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. *The S2E Platform: Design, Implementation, and Applications*. Ecole Polytechnique Federale de Lausanne (EPFL), 2012. To appear in ACM Transactions on Computer Systems, 2012.

[Com08] Python Community. *Python 2.5.4*. Python Software Foundation, 2008. http://www.python.org/getit/releases/2.5.4/.

[CSXK08] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSID 2008)*, pages 255–266, December 2008.

[CW12] Vitaly Chipounov and Jonas Wagner. *S2E Developer Forum*. S2E Developer Forum, 2012. http://groups.google.com/group/s2e-dev?hl=en&src=email.

[dMB09] Leonardo de Moura and Nikolaj Bjorner. Satisfiability Modulo Theories: An Appetizer. In *Lecture Notes in Computer Science*, pages 23–36, 2009.

[Eag08] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2008.

[Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005.

[FCN^{+}10] A. Franzen, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev. Applying SMT in Symbolic Execution of Microcode. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 121–128, Oct. 2010.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[GFcC08] Fanglu Guo, Peter Ferrie, and Tzi cker Chiueh. A Study of the Packer Problem and its Solutions. In *RAID 2008, LNCS 5230*, pages 98–115, 2008.

[HM79] R.N. Horspool and N. Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal, Vol 23 No 3*, pages 223–229, 1979.

[HR11] Hex-Rays. *The IDA Pro Diassembler and Debugger*. Hex-Rays, Blvd de la Sauvenire 30, 4000 Lige, Belgium, June 2011. http://hex-rays.com/idapro/.

[KHL10] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *19th IEEE Asian Test Symposium*, pages 59–64, 2010.

[KLMK10] Jae-Jin Kim, Seok-Young Lee, Soo-Mook Moon, and Suhyun Kim. Comparison of LLVM and GCC on the ARM Platform. In *Embedded and Multimedia Computing (EMC)*, pages 1–6, 2010.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[LAB11] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*, pages 251–268, Feb 2011.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[Lat11] Chris Lattner. *The LLVM Compiler Infrastructure*. Computer Science Department at University of Illinois at Urbana-Champaign, April 2011. http://llvm.org/.

[Ltd12] ARM Ltd. *The ARM Instruction Set - ARM University Program - v1.0*. ARM Ltd., 2012. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html.

[LZX10] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Network and Distributed System Security Symposium*, pages 1–20, 2010.

[OMR12] Markus F. X. J. Oberhumer, Laszlo Molnar, and John F. Reiser. *Ultimate Packer for eXecutables*. SourceForge.net, 2012. http://upx.sourceforge.net.

[Res11] Grammatech Research. *CodeSurfer/x86*. Grammatech Research, 2011. http://www.grammatech.com/research/products/CodeSurferx86.html.

[Sco11] Jeffrey B. Scott. Automated Analysis of ARM Binaries Using the Low-Level Virtual Machine Compiler Framework. Master's thesis, Air Force Institute of Technology, 2011.

[Seb99] Robert W. Sebesta. *Concepts of Programming Languages, 4th ed.* Addison-Wesley Longman, Inc., 1999.

[Sou12] Open Source. *IDAPython - Python Plugin for Interactive Disassembler Pro*. Google Code, 2012. http://code.google.com/p/idapython/.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, pages 1–20, 2011.

[Yus11] Oleh Yuschuk. *The OllyDbg Debugger*. Oleh Yuschuk, 2011. http://www.ollydbg.de.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 22-03-2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* Aug 2010- Mar 2012

4. TITLE AND SUBTITLE
Binary Disassembly Block Coverage by Symbolic Execution vs. Recursive Descent

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Miller, Jonathan D. 2d Lt USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCO/ENG/12-09

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

This research determines how appropriate symbolic execution is (given its current implementation) for binary analysis by measuring how much of an executable symbolic execution allows an analyst to reason about. Using the S2E Selective Symbolic Execution Engine with a built-in constraint solver (KLEE), this research measures the effectiveness of S2E on a sample of 27 Debian Linux binaries as compared to a traditional static disassembly tool, IDA Pro. Disassembly code coverage and path exploration is used as a metric for determining success. This research also explores the effectiveness of symbolic execution on packed or obfuscated samples of the same binaries to generate a model-based evaluation of success for techniques commonly employed by malware. Obfuscated results were much higher than expected, which lead to the discovery that S2E was not actually handling the multiple executable memory regions present in unpacker runtime code. Three recommendations are made to address the shortcomings of S2E and allow it to process obfuscated samples correctly.

15. SUBJECT TERMS
Disassembly, Symbolic Execution, Recursive Descent, Obfuscation, Binary Analysis

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Rusty O. Baldwin
REPORT U	**ABSTRACT** U	**c. THIS PAGE** U	UU	115	**19b. TELEPHONE NUMBER** *(Include area code)* 937-255-6565 ext. 4445 rusty.baldwin@afit.edu